

LEVERAGING MARL TECHNIQUES TO SPEED UP LEARNING BY DECOMPOSING BIG ACTION SPACES

JOAQUIM MARSET ALSINA

Thesis supervisor: MARIO MARTÍN MUÑOZ (Department of Computer Science)

Degree: Master Degree in Artificial Intelligence

Master's thesis

School of Engineering
Universitat Rovira i Virgili (URV)

Faculty of Mathematics
Universitat de Barcelona (UB)

Barcelona School of Informatics (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

20/10/2023

Acknowledgements

First, I want to thank Professor Mario Martín, the thesis supervisor, for the thesis idea and all the support, help, and advice. He was the one who introduced me to the area of multi-agent reinforcement learning (MARL), the topic of the project, and I consider him one of the best professors I have had in this master's degree.

Second, I want to thank Matteo Gallici and Bartomeu Poumulet, two PhD students working with Mario who helped me with two contributions we wanted to make but could not finish. Nonetheless, I want to thank both for their advice and help.

Third, I want to thank Dr Ulises Cortés, this master's coordinator, and the Barcelona Supercomputing Center (BSC) for granting me access to computer resources to facilitate the project fulfilment. Besides, I want to thank the BSC's support staff for all the help in setting up the environment.

Finally, I want to thank my family and friends for all the support and encouragement. I do not forget to thank all the people I have met during this master's degree as they have livened up and eased this journey.

Abstract

This thesis explores the field of multi-agent reinforcement learning (MARL). It presents a solution to tackle the problems single-agent methods face when they cannot solve a task because the action dimensionality is too high. As they train one network to output an action, if that action has too many dimensions, it needs to explore exponential possibilities to find the optimal one to execute at each state of the environment.

For this reason, we propose dividing those actions among multiple agents, each controlling one or more dimensions. Thus, we can train several networks to output the sub-action dimensions each agent controls, each having to explore fewer combinations. With that solution, we have studied different MARL methods to demonstrate how they can solve those tasks that single-agent methods cannot. We have also studied which type of MARL method works better.

We divide a single-agent task into multiple agents but have a shared reward function that we cannot usually decompose into each agent's reward function. Hence, different methods tackle the problem of learning each agent's policy with a shared reward that depends on the actions of all agents. In this thesis, we have focused on those learning to decompose that shared reward into the contributions of each agent.

Thus, we have studied several MARL methods belonging to that type and demonstrated how they work better than those not decomposing the joint reward and those that train by ignoring the existence of the other agents, even though their actions affect the environment.

Finally, we have focused on methods working with continuous actions because there is less research. Thanks to finding very few, we have developed a new one. We have developed that method using another as the basis because there were some points about it we thought we could improve by doing some modifications. Thus, we have also compared our new method with the others, obtaining the best results in most cases.

Contents

Summary of Notation

We have used two notations in this thesis: the one from [39] when explaining the concepts of single-agent RL and the one from the different papers when explaining the concepts of multi-agent RL. They mostly do not overlap, and the notation from the articles is not that different. Nonetheless, we summarize the most used below:

t	discrete time step
s, s', s_t	states
a, a', a_t	actions
r, r'	rewards
$\pi(s)$	deterministic policy representing the action taken in state s
$\pi(a\|s)$	stochastic policy representing the probability of taking action a in state s
G_t	discounted return obtained from step t onwards
S_t	random variable representing the state the agent is in step t
A_t	random variable representing the action the agent executes in step t
R_t	random variable representing the reward the agent receives in step t
$v_\pi(s)$	state-value function of policy π evaluated on state s
$q_\pi(s, a)$	action-value function of policy π evaluated on state-action pair (s, a)
$V(s, w)$	approximation of $v_\pi(s)$ using a parameterized function with weights w
$Q(s, a, \phi)$	approximation of $q_\pi(s, a)$ using a parameterized function with weights ϕ
$\pi(s, \theta)$	approximation of $\pi(s)$ using a parameterized function with weights θ
$\pi(a\|s, \theta)$	approximation of $\pi(a\|s)$ using a parameterized function with weights θ
ϕ_{targ}, θ_{targ}	outdated vector of weights used in target networks
γ	discount factor used to compute G_t
ρ	scalar weight used in polyak average to update the target weights
o_a, o^a	observation of agent a
u_a, u^a	action of agent a
$\boldsymbol{u}$	joint action of all agents (i.e. vector with the actions of all agents)
$\boldsymbol{\tau}$	trajectory containing the observations and actions of all agents
τ^a, τ_a	trajectory containing the observations and actions of agent a
$\pi(\boldsymbol{u}\|\boldsymbol{\tau})$, $\pi(\boldsymbol{\tau})$	joint policy (i.e. decides an action for all agents)
$\pi_a(u_a\|\tau_a)$, $\pi_a(\tau_a)$	policy of agent a
$q_\pi(\boldsymbol{\tau}, \boldsymbol{u})$, $q_\pi(s, \boldsymbol{u})$	joint action-value function
$Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}, \theta)$, $Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}, s, \theta)$	approximation of the joint action-value function
$Q_a(\tau_a, u_a, \theta_a)$	utility function representing agent's a contribution to the joint reward (They are not real action-value functions)

1 Introduction

1.1 Motivation

We can use reinforcement learning (RL) to solve many real-world problems: a robotic arm grabbing a ball from a table or even an autonomously driven car. As an introduction to what comes in section 2, RL considers a single agent (or a group of agents) acting in an environment to achieve a certain goal (e.g. driving the car to a certain destination).

The environment has states (e.g. the ball and arm position), and when the agent acts, the environment transitions to a new state and gives a reward signal as feedback. This reward informs the agent about its progress in achieving the goal as a way to correct the agent's actions. Hence, an RL method learns the proper behaviour to select the correct action at the given state to reach the agent's objective.

RL uses neural networks to learn the agent's behaviour. For example, we can train a network to output the correct action given the environment state as input. Like in a multi-class classifier in supervised learning, we generate a probability distribution of the actions the agent can take and sample from that probability. Ideally, we should modify the network's weights to shift the probability mass to the action giving the highest accumulated reward, moving the agent to the desired goal.

If each action has N dimensions and each dimension can take n_d values, the network generates n_d^N probability values. It is feasible in benchmarked and simulated tasks as we usually have few low-dimensional discrete actions (i.e. each dimension a finite set of values). However, in real-world problems, where it is not strange to have high-dimensional continuous actions (i.e. each dimension an interval), we cannot maintain this idea of a single "brain" (i.e. network) controlling the agent's behaviour. Not only because of memory requirements but because it hinders learning.

Therefore, in more complex environments where a single "brain" is not feasible, we have better options to speed up and improve learning. In subsection 3.1, we will present some options, which are not the main focus of this thesis, that keep a single network but perform certain tricks and modifications in the architecture to avoid exploring all the combinatorial action space. Nevertheless, these methods mostly work on simpler discrete cases. Our main focus is taking advantage of multi-agent RL (MARL) to divide the action space among a set of agents, each exploring a subset of the action space.

In MARL, we consider multiple agents interacting with the environment, each with its actions set. The agents do not observe the environment state, but some limited and local information to them we refer to as observation. Finally, the agents can work cooperatively to reach a shared goal or even compete if they have conflicting goals.

As we want to divide an initially single-agent environment into multiple agents to maximize the original reward and achieve the same objective, we will always consider cooperative MARL approaches. Hence, several agents will share the original reward function they try to maximize by acting cooperatively. We usually assume a shared reward because it is difficult to divide a cooperative goal into the goals of each agent.

To divide a single-agent problem into a multi-agent system, we must perform some transformations and meet some requirements. First, we need an action structure we can divide between the different agents. The simplest case is when we have multi-dimensional actions and assign one or more dimensions to an agent. These dimensions do not need to be independent but can be correlated. In that case, we need to sample them sequentially and ensure the agents can communicate to inform about the action(s) each one depends on.

Second, we must divide the environment state into the agents' observations. The state contains the information of all the environment's entities, including the agents and any other entity that could exist. In MARL, the agents have limited observability and cannot observe the global state but only information about themselves (e.g. their position). In some environments, the agent can also obtain other entities' information if they are in its view radius or through communication. Nonetheless, in most cases, the agents make decisions based only on their information (i.e. act decentralized).

Third, we need to have a shared and global reward function, which is more of a requirement for MARL methods solving coordination tasks than a necessary condition for all methods solving multi-agent tasks. Moreover, designing a specific reward function for each agent is difficult. Considering how we start from a single-agent problem, this requirement is easy to fulfil.

If we meet these conditions, we can transform the problem into a multi-agent system and use the state-of-the-art MARL approaches to solve it. We move from an agent exploring a combinatorial action space with n_d^N actions to N agents, each exploring n_d actions. Nonetheless, switching to MARL introduces some challenges like learning cooperation and credit assignment, which, in some cases, complicate learning the task even more than if we tackle it as a single-agent problem.

A second advantage of leveraging MARL is training smaller networks. The networks usually receive the agent's observation as input and output the probability of the agent's actions (i.e. independent of the number of agents). Moreover, we do not need a network per agent, but we can train a single network to output actions for each agent, even if they do not behave equally but have the same input and output size. It is even better if they identically behave because we can reuse this learned behaviour with as many agents as we want, allowing us to scale the same task to more agents.

A third advantage (we could not prove it) is learning general policies to reuse in similar tasks. Imagine an environment featuring a bipedal robot learning to walk to the right. We could train two agents, each controlling one leg. The robot is symmetric, so both legs should behave identically but probably act at different times, like we humans walk. If we can learn an optimal behaviour to walk in one direction, we could "rotate" the environment 180^{o} and make it walk to the left instead. A single "brain" would have a harder time, having to learn each task independently.

Among MARL methods, we want to focus on those based on value function factorization, explained later in detail. These methods tackle the problem of the agents sharing a reward by learning to decompose it into the contribution of each agent to that reward. It helps to learn in those environments where an agent needs to sacrifice itself for the greater good as the shared reward fails to encourage the agent. Nonetheless, we will also try methods that do not decompose the value function to make comparisons and determine which works better.

Finally, we also want to focus on environments with continuous actions. Most RL research, including MARL, focuses on discrete ones because we can always discretize an interval and tackle the problem as discrete. However, there are problems where we might need the more fine-grained control the continuous actions offer. Even though the most used MARL benchmarks (e.g. StarCraft II micro-management task [35]) have discrete ones, real-world scenarios have continuous ones, and we should focus on them.

1.2 Contributions

We have studied several MARL methods to demonstrate how one can decompose a single-agent environment into multiple agents and leverage those methods to solve tasks that a single-agent method could not. We have tested different environments with each method, comparing their results.

We have searched for environments we could make more complex by increasing the action dimen-

sionality (i.e. the number of agents) and the agents behaving identically. MARL methods can learn to solve a task and then reuse it in a more complex version if we train a behaviour all agents share. We have demonstrated how, without further training, we can achieve almost optimal performance or use it as a starting point to fine-tune for fewer steps than training from scratch. On the other hand, a single-agent method cannot reuse any behaviour and needs to learn each task from scratch, sometimes failing to learn the smaller task.

We have focused on MARL methods decomposing the joint reward into the different agents' contribution, demonstrating, in the different environments, how they work better than those not performing this decomposition and using the joint reward to learn the agent behaviours.

We have prioritized experimenting with environments having continuous actions. Most state-of-the-art MARL research develops methods to solve benchmark tasks with discrete ones. However, real-world problems usually have continuous actions, and we should focus on developing methods that could solve them. As we will later see, we did not find a lot of existing MARL methods working with continuous actions.

We intended to demonstrate how MARL methods can learn general policies that can solve similar tasks compared to a single-agent method that needs to solve each task separately. However, we could not reach conclusive results even though we still believe they could learn such policies. Thus, we cannot count it as a contribution.

Finally, we have developed a new MARL method working with continuous actions and decomposing the joint reward, using another we will later explain as a basis, improving its results by performing some modifications and improvements.

1.3 Overview

We have organized this thesis as follows. In section 2, we provide a theoretical background with the topics required to understand this work. We start with the single-agent case and then extend to the multi-agent setting, which shares all concepts. In particular:

- subsection 2.1 introduces the most basic RL concepts, like the agent, policy, and reward.
- subsection 2.2, subsection 2.3, and subsection 2.4 introduce the three types of RL methods: value-based, policy gradient, and actor-critic.
- subsection 2.5 explains how to implement those methods using deep learning, presenting examples of the most well-known and with state-of-the-art performance.
- subsection 2.6 explains the new concepts and changes we need to perform to extend to MARL, the challenges of having multiple agents, and the most common way to train MARL methods to solve some of those challenges.

In section 3, we perform a literature review of methods dealing with large action spaces. We start in subsection 3.1 with those keeping a single agent but performing different tricks to make the problem tractable. Afterwards, in subsection 3.2, we present the most well-known and state-of-the-art multi-agent RL methods, including those based on value function factorization and those that do not, both working with discrete or continuous actions.

In section 4, we explain the different environments used in our experiments, including the modifications and adaptations to make them work with multiple agents. As we want to demonstrate that we can speed up training using multiple agents, most are originally single-agent.

In section 5, we better explain the existing MARL methods we compare in our experiments, which we have introduced in subsection 3.2. In section 6, we explain the new method we have developed using one of the previous as the basis.

In section 7, we explain the experiments we performed using the different environments and methods and discuss their results. We started with one working with discrete actions to familiarize ourselves with MARL and then moved to those working with continuous actions, our main focus. Nonetheless, we still use the first one to demonstrate the advantages of MARL methods.

In section 8, we discuss our findings and the analysis of the results we performed in the experimentation, objectively describing the degree of fulfilment of the thesis' goals. We wrap up the section by proposing possible lines of future work.

We end with some appendices. First, in Appendix A, we explain the hyperparameters the different RL methods use. Second, in Appendix B, we show frames of some episodes we rendered to view how the methods learned to solve the environments. Finally, in Appendix C, we explain what one can find in our GitHub repository.

2 Background

2.1 Reinforcement Learning

Reinforcement learning (RL) is a sub-field of machine learning (ML) featuring an agent interacting with an environment to achieve a particular goal. The agent interacts by doing actions that modify the environment state, and the environment returns the progress to the agent's goal in the form of a scalar reward. The goal of RL is learning a behaviour to decide the optimal action to execute at each environment state such that the agent maximizes the accumulated reward.

The agent learns by trial and error, one way we humans learn. We do not know the best answer from the beginning, but by acting and experiencing the consequences, we refine our actions. Imagine yourself trying to bake a cake. The first time, you will probably follow a recipe that suits you. However, the result will probably not be what you expected. Maybe, it is not sweet enough, or you have not cooked it enough. The next time, you will fix it by adding more sugar or cooking it for more time. After multiple tries, you will have mastered the recipe and even improved it.

Learning from interaction is very different from supervised and unsupervised learning. Supervised learning always has the correct answer, and the model only learns to reproduce it and generalize on unseen situations. If the action is the incorrect answer, it does not care about how good it was and its consequences. RL is also different from unsupervised learning, as the latter tries to extract hidden information from the data instead of using it to maximize a reward signal. Both do not rely on labelled data, but their goals are different.

The interaction between the agent and the environment is continuous, at the level of discrete time steps, and is usually represented as in Figure 1:

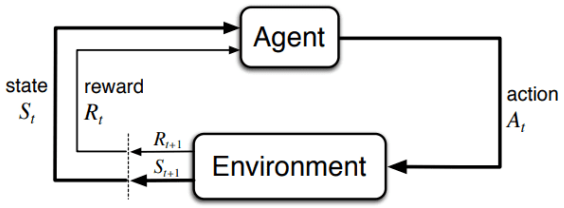


Figure 1: RL framework

At each step t, the agent observes the state S_t of the environment and produces an action A_t based on that state. This action modifies the environment, causing a transition to a new state S_{t+1}, and a reward R_{t+1} to enter that new state. This process continues until the environment reaches its final state or never ends if it has infinite states. We call trajectory the sequence of states, actions, and rewards we encounter until the episode ends or until infinity, denoted as τ.

Imagine, for example, an RL problem featuring a person (i.e. the agent) trying to escape a maze. The environment state could be what the person sees, the person performing actions like moving forward, turning left, or turning right. From those actions, the environment state would change as the person has moved or rotated, and the environment could reward the agent for how much it has advanced to the exit since the last action.

We usually formalize RL problems as a Markov decision process (MDP) [3, 39], which provides a framework for modelling decision-making in situations where the outcomes are partly random and partly controlled by a decision-maker. Nonetheless, in RL, we assume the environment only changes due to the agent's actions, and we do not model any other entity (e.g. another agent) that could change it. We define an MDP as the tuple $< S, A, P, R >$:

- S is the set of environment states.
- A is the set of agent actions.
- $P : S \times S \times A \to [0, 1]$ is the transition function, a probability distribution over the next states. If we are in state s and take action a, $P(s'|s, a)$ defines the probability of reaching state s'. This function, which we also call the dynamics of the environment, is usually unknown.
- $R : S \times S \times A \to \mathbb{R}$ is the reward function. Defined as $R(s, a, s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s']$, determines the expected reward the agent can obtain for executing action a in state s and reaching the next state s'. We consider an expectation given that the same transition can produce different rewards. This function is also unknown in most cases.

An interesting remark is that both the probability and the reward function depend only on the previous state and action and forget all the previous transitions. This is because the state in an MDP follows the Markov property, which says that a state must contain all the information of the previous transitions. Thus, we can forget about all the past state-action pairs and only condition those functions on the current state and action.

A second important concept in an RL problem is the return G_t, the sum of rewards obtained from step t onwards. Given that the environment may not have a final state, and we would end up with an infinite sum, we always compute a weighted version where we weigh each term with a weight $\gamma \in [0, 1)$ that we call discount factor. Hence, the discounted return at step t is $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. As bigger is γ, we take more into account the future steps.

A third concept is the policy π, a function that maps states to actions and the one deciding which to take at each time step. The policy can be deterministic or stochastic. The former maps each state to one action (i.e. $\pi(s) = a$), and the latter maps each state to a probability distribution (i.e. $\pi(a|s) = P(A_t = a|S_t = s)$). Solving an MDP (and the goal of RL) is learning the optimal policy π^* that maximizes the expected return from the initial state. In particular:

$$\pi^* = argmax_\pi \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t, S_{t+1})] \tag{1}$$

To end up, we want to explain two more concepts, the state-value function $v_\pi(s)$ and the action-value function $q_\pi(s, a)$. The former determines how good being in state s is, and the latter how good being in state s and executing action a is. We define the state-value function $v_\pi(s)$ as the expected return an agent can obtain starting in state s and following the policy π:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']], \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')], \ \forall s \in S \end{aligned} \tag{2}$$

We can see how the state-value function represents the goal of the RL problem. Hence, most algorithms maximize v_π (or q_π) at each state to obtain the optimal policy. The last equality in Equation 2 is called the Bellman equation [39], expressing the relationship between $v_\pi(s)$ and $v_\pi(s')$. We can only compute it exactly if we know the environment's dynamics p (i.e. the transition function), which is not usually the case. For this reason, most algorithms approximate the state-value function using different methods (e.g. with a neural network).

We define the action-value function $q_\pi(s, a)$ as the expected return an agent can obtain starting in state s, taking action a, and then following the policy π:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a], \ \forall s \in S, \ a \in A(s) \end{aligned} \tag{3}$$

Like the state-value function, we can also develop a Bellman equation for the action-value function if we remove the outer expectation and write it in terms of the transition function. Therefore, in most cases, we cannot either compute it exactly, and we need to approximate it.

We can approximate both leveraging Monte Carlo methods [39], which estimate an expectation by obtaining multiple samples and computing an average. For example, we can approximate the action-value function by running several episodes and computing the discounted return for each state-action pair. We will visit the same pair several times, obtaining multiple returns with each to calculate an average. The problem with Monte Carlo methods is that we need to wait until finishing an episode to compute the returns.

2.2 Value-based Methods

Value-based methods find the best policy without computing it explicitly, but they obtain it using the action-value function. As we have said, we do not have the environment's dynamic function, so we need to approximate the action values. We refer to those methods that do not rely on the environment's dynamics as model-free methods.

The most important method is Q-learning. It approximates the action-value function for each state-action pair by iteratively refining an initial estimate using an incremental approach until convergence. In particular, it computes a new value using the old one and the difference with the target value we want to reach (i.e. $new\_estimator = old\_estimator + step\_size * (target - old\_estimator)$). We store all the action values in a tabular structure, explaining why we can only use it with a finite (i.e. discrete) and small enough set of states and actions. We compute the estimates as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{4}$$

From these estimates, we can obtain the action a to execute in a particular state s by selecting the one with the highest action value (i.e. the greedy one), which will be the one π^* would output for s. In particular, with Q-learning, we obtain the policy as follows, which might not be optimal if we do not estimate the values until convergence:

$$\pi(s) = argmax_a Q(s, a) \tag{5}$$

That is why we say we do not learn the policy explicitly, as we only learn to estimate the action-value function and compute an *argmax* operation. The problem with this method is that we exhaustively search for all the actions to select the one with the highest action value, which limits the applicability to a finite set of actions.

We do not estimate the policy with the state-value function, as it depends only on the state but not the action. To include the action, as we have seen in the Bellman equation of $v_\pi(s)$, we need the transition probability, which we do not usually have. That is why most value-based methods use the action-value function instead, which makes the action selection process easier.

It is not the case with Q-learning, but imagine a value-based algorithm iteratively approximating q_π and always selecting the greedy action. What will happen is that it will leave a lot of state-action pairs unexplored because the action it believes is the best at the early iterations is not the best in the long run because it had inaccurate approximates of q_π at that moment and never explored other options. Thus, it will learn to select the wrong action in most states, resulting in a sub-optimal policy and return.

To solve this problem, we must introduce some form of exploration that selects sub-optimal actions to improve the action-value function approximation. However, we cannot keep this exploration forever, as we still want to select the greedy action that will produce the highest return. This conflict is known as the exploration and exploitation trade-off. We must balance selecting the greedy action (exploitation) and exploring other options that do not seem the best at first sight (exploration).

The exploration method used in Q-learning is ϵ-greedy exploration. The idea is that with probability $1 - \epsilon$, we select the greedy action, and with probability ϵ, we select one action at random (i.e. each action has probability $\frac{\epsilon}{|A|}$). We only use this exploration when interacting with the environment. However, we use the greedy action when computing the target value to update the action-value estimates.

This introduces another new concept: the difference between on-policy and off-policy methods. In Q-learning, we use two policies, one to interact (ϵ-greedy) and another to update the estimates (greedy), making the method off-policy. On-policy methods use the same policy to interact and update the estimates.

The exploration should decrease with time, as when we better approximate the action values, computing an *argmax* should give us the current best action for that state. However, it should never disappear, as we could still be selecting a good but not optimal action because the approximates have not converged yet. For example, the algorithm could have two actions with the highest action value, only one being optimal. Thus, the value of ϵ usually starts at 1.0 and decreases until a minimum value higher than 0 (e.g. 0.05).

Storing all the estimates for each state-action pair is feasible if we have a reasonable amount. Moreover, computing an *argmax* over all actions is only possible if they are discrete (i.e. a finite set). Finally, Q-learning needs to visit a pair to update its approximation, which can be impossible to visit each enough times if both state and action space are huge. Therefore, if we have continuous actions (i.e. a real-valued interval) or our states are, for example, large images, Q-learning is intractable in both time and space.

For this reason, there is another set of methods that approximate the state-value (or action-value) function, using some parametric function with weight vector w (i.e. $V(s, w) \approx v_\pi(s)$). We need to find the weights w that best approximate v_π using methods like gradient descent. One example could be using a linear function $V(s, w) = w^T x(s)$ (being $x(s)$ a state representation). Another approximation could be via deep neural networks, which we will later see.

Independently of the parametric function, these other methods only need to store the weight vector and can visit only a subset of pairs because updating the weights for one pair also affects others (i.e. generalizes to unseen ones). Nonetheless, they still need to explore enough of the action space, like Q-learning, to avoid reaching sub-optimal policy.

2.3 Policy Gradient Methods

Policy gradient methods learn an approximation of the policy using a parameterized function, $\pi(a|s, \theta)$, to select actions without consulting a value function. We can still use a value function to learn the policy parameters θ, but we do not use it for action selection. From this point onwards, we do not consider tabular approaches and always use function approximation to learn both the policy and the value functions. Compared to value-based, there are several advantages of using policy gradient methods:

- They can learn stochastic policies, which value-based cannot. Learning stochastic policies has some advantages on its own:
  - They do not need an explicit exploration mechanism, as they select actions sampling from a probability distribution.
  - The agent does not suffer from the problem of aliased states, in which two appear to be the same, but the agent needs different actions in each state. In that case, a deterministic policy can only learn one, and the agent might not reach the goal state.
- They can handle continuous action spaces because to select the optimal action in one state, they do not need to compute a maximum over all the action values, which is impractical with large action spaces.
- They have better convergence properties, as they directly optimize the policy parameters and do not depend on the action values. In value-based methods, we can suffer from high oscillations during training because a small change in the action values can change the selected action.

However, they also have some disadvantages. Policy gradient methods always converge, but a lot of times, to a local optimum rather than the global one. Besides, these methods converge slower than

value-based ones and usually take longer to train.

We want to learn the policy parameters θ that result in the best policy. For this reason, we need some function $J(\theta)$ to evaluate the quality of the current policy, which is also the function we want to maximize by changing those parameters. One option is the state value of the initial state: $J(\theta) = v_\pi(s_0)$. We can use gradient ascent and back-propagation, meaning that at each step t, we compute the gradients of $J(\theta_t)$ to optimize θ_t:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{6}$$

Without going into detail, computing the exact gradients is difficult, as we depend on the unknown state distribution. However, thanks to the policy gradient theorem [39], we can obtain an exact gradient expression in the form of an expectation, and we can sample it to update the parameters. Therefore, we do not compute the gradient to update the parameters, but a sample whose expectation equals the gradient. In particular, the gradient expression is as follows:

$$\nabla J(\theta) = \mathbb{E}_\pi[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \theta)] \tag{7}$$

The first and most important policy gradient algorithm is called REINFORCE [51, 39], which starts with the previous gradient equation and further derives it by introducing the random variable A_t, replacing the sum under a (i.e. under the possible values of A_t) with an expectation under π, and then replacing q_π for G_t (i.e. q_π is already an expectation under π). Like before, we sample the expression to compute the gradients. In particular:

$$\begin{aligned}
\nabla J(\theta) &= \mathbb{E}_\pi[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \theta)] \\
&= \mathbb{E}_\pi[\sum_a \pi(a|S_t\theta) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \theta)}{\pi(a|S_t, \theta)}] \\
&= \mathbb{E}_\pi[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}] \\
&= \mathbb{E}_\pi[G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}]
\end{aligned} \tag{8}$$

With this expression, we do not need to know the action values of all the actions in the state S_t, but we only need one, the action we select. Thus, by simply interacting with the environment and selecting actions, we can update θ_t by sampling the previous expectation. The resulting gradient ascent equation is the following:

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \\
&= \theta_t + \alpha G_t \nabla ln(\pi(A_t|S_t, \theta))
\end{aligned} \tag{9}$$

The gradient of π represents the direction in parameter space that most increases the probability of taking action A_t in state S_t. We update θ_t in the gradient direction proportional to the return and inversely proportional to the probability. Therefore, we update the parameters such that we select actions with the highest return more frequently but also favour exploration and select those with the highest probability less often, given that they might not yield the highest return.

REINFORCE uses the complete return from step t, G_t, which includes all the rewards until the end of the episode (i.e. we can consider it a Monte Carlo approach). This means we must wait until the end of the episode to update the parameters. Moreover, G_t will greatly differ between episodes, causing highly-variant gradients and making learning unstable.

We can solve this instability by subtracting a baseline $b(S_t)$ from G_t to reduce the magnitude of the gradients and make more stable updates. We can use any function as long as it does not vary with the

action to keep the expected value of the update unchanged but reduce its variance. Typically, we use an estimate of the state value, $V(S_{t+1}, w)$, being w, another set of learnable weights. The new weight update equation is as follows:

$$\theta_{t+1} = \theta_t + \alpha(G_t - V(S_t, w_t))\nabla ln(\pi(A_t|S_t, \theta)) \tag{10}$$

To solve the problem of waiting until the end of the episode, we can replace the return for some equivalent quantity like the state-value function. We have seen in Equation 2 how the Bellman equation defines the value of a state in terms of its successor states. If we rewrite it with the expectation as follows:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{11}$$

We only need to perform one step in the environment and then approximate the value of $v_\pi(S_{t+1})$ (i.e. $V(S_{t+1})$), but we do not need to wait until the end of the episode to compute the return and perform a policy update. We call this type of method Temporal Difference (TD), this one, in particular, one-step TD. The new update equation is as follows:

$$\theta_{t+1} = \theta_t + \alpha(R_t + \gamma V(S_{t+1}, w) - V(S_t, w))\nabla ln(\pi(A_t|S_t, \theta)) \tag{12}$$

With that, we only need some way of approximating the state-value function (e.g. using neural networks). We call the expression $R_t + \gamma V(S_{t+1}, w) - V(S_t, w)$ one-step TD error, and we will see it a lot of times in this document, as a lot of advanced RL methods use it in their update equations.

Another advantage of leveraging the one-step TD equation is evaluating each action independently instead of the whole set that moved from state S_t to the final one. Even if G_t is high, it does not mean all the executed actions were equally good. Maybe some sub-optimal action will become more probable because G_t was high. However, if we use the one-step TD method, we update the gradients after each transition with the environment, modifying each action probability independently.

2.4 Actor-critic Methods

Actor-critic methods learn to approximate the policy and the state-value (or action-value) function using some parameterized function with parameters θ and w, respectively. The actor learns the policy and the critic the value function. We call them actor and critic because the actor acts in the environment, and the critic criticizes the actor's actions. In Figure 2, we present the interaction of the actor, the critic, and the environment:

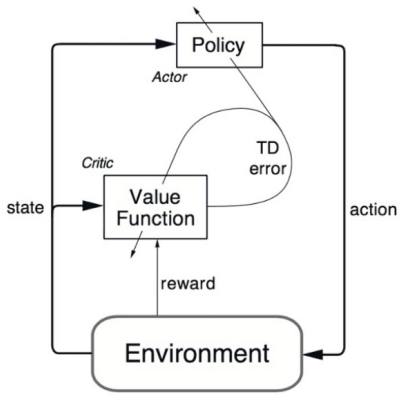


Figure 2: Actor-critic framework

The critic gives feedback to the actor through the term $V(S_{t+1}, w)$ present in the actor update equation. If the action is not good, the value of S_{t+1} will be low, and we will decrease its probability.

Actor-critic extends policy gradient, meaning we can update θ using Equation 12. Nonetheless, we want to mention that the baseline term $V(S_t, w)$ does not provide any feedback and is only used to reduce the variance. For this reason, the method that used G_t cannot be considered actor-critic.

We must update the critic's parameters to improve the value function approximation. If not, the critic cannot guide the actor correctly. We can use the following update equation, where δ is the one-step TD error, which we want to reduce to make the current estimate $V(S_t, w)$ more similar to the target $R_{t+1} + \gamma V(S_{t+1}, w)$:

$$\begin{aligned} w_{t+1} &= w_t + \alpha \delta \nabla V(S_t, w) \\ \delta &= R_{t+1} + \gamma V(S_{t+1}, w) - V(S_t, w) \end{aligned} \tag{13}$$

Thus, we perform gradient descent instead of gradient ascent. $\nabla V(S_t, w)$ represents the direction in parameter space that most increases the value of state S_t. We want to update the parameters to make the value bigger when the error is positive because we are underestimating $V(S_t)$. And we want to reduce it when the error is negative because we are overestimating $V(S_t)$.

2.5 Deep Reinforcement Learning

Deep reinforcement learning (DRL) uses deep neural networks to approximate the policy and the value functions. They can easily handle environments with large state spaces like images or text and large continuous action spaces. Moreover, we can use them to implement value-based, policy gradient, and actor-critic methods.

To approximate the policy, we use different network architectures depending on the policy type and, in the case of stochastic, the action type. In Figure 3, we show an example of each architecture, all receiving the state as input. Deterministic policies' architecture directly outputs the action, while stochastic's outputs the distribution we sample from. Categorical when having discrete actions (i.e. each action's probability), and Gaussian when continuous ones (i.e. its parameters).

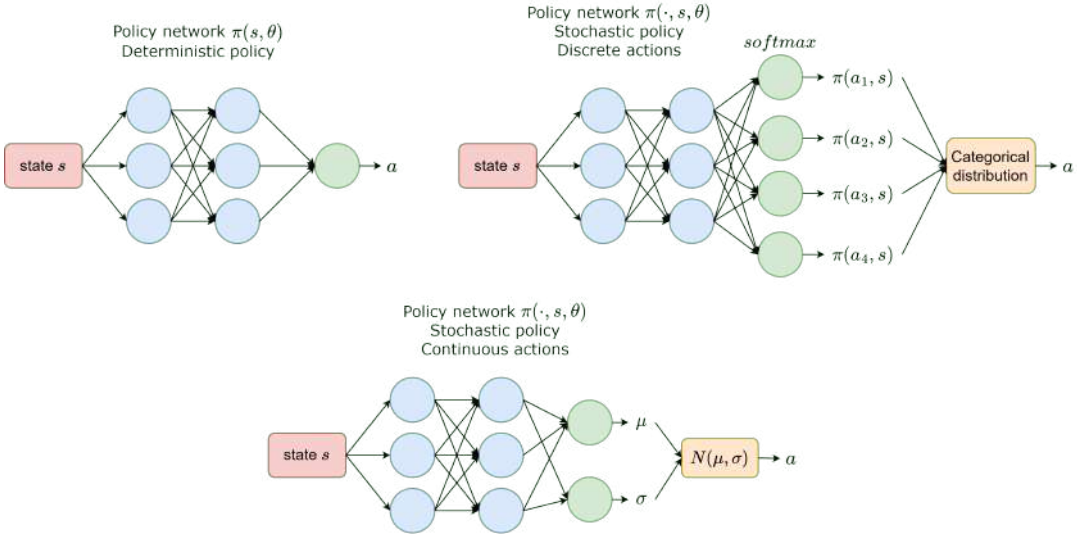


Figure 3: Different types of policy network

To approximate the state-value function, we feed the state as input and directly output the state value. In Figure 4, we present an example of a state-value network.

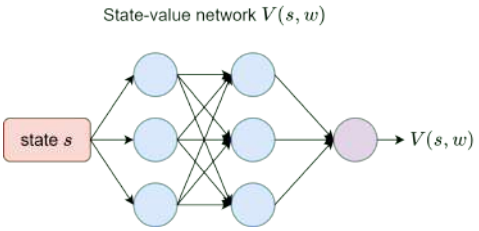


Figure 4: State-value network

To approximate the action-value function, we also distinguish the type of action. In the discrete case, we feed the state and output all the action values of that state. In the continuous case, we input both the state and the action and output their action value. In Figure 5, we show an example of both types of networks.

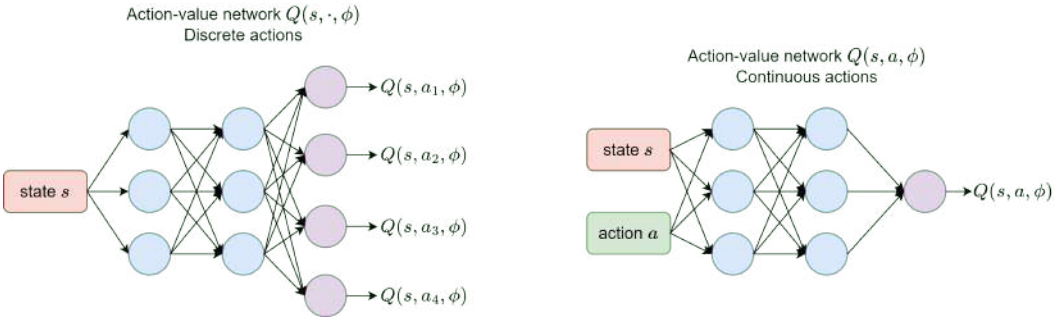


Figure 5: Different types of action-value network

In the previous examples, we have shown fully connected networks, as we have not entered into details of the state structure. We could have images as states, suggesting using convolutional neural networks (CNN). Or we could have text, and using recurrent neural networks (RNN) could be a better option. Nevertheless, each algorithm defines its most suited architecture.

As we do in supervised learning to update the networks, we need some function to compute the gradients for the network parameters using back-propagation and perform gradient ascent or descent. Depending on the type of network, we want to maximize or minimize such function. When updating the action-value networks, we want to minimize a loss function, which we denote with the letter L. However, when we update the policy networks, we want to maximize an objective function (i.e. the RL goal), which we specify with the letter J.

The function $J(\theta)$ we presented before was an example of a function to maximize when updating the policy network. Nonetheless, most deep learning frameworks (e.g. Pytorch) only allow to perform gradient descent. For this reason, when looking at some method implementation, we will always see the negated version of $J(\theta)$, which they also refer to as a loss function because it is the same minimizing the negated version as maximizing the positive one.

Below, we briefly explain several single-agent DRL algorithms, including those we have used in this thesis and some we consider major to understand other methods and concepts we will later present.

- **Deep Q-Networks**:
  Deep Q-Networks (DQN) [27, 26] is the most famous DRL algorithm. It is an off-policy value-based method used in environments with discrete action spaces. It cannot handle continuous

actions because it needs to evaluate a maximum over all possible actions when evaluating and improving the policy.

All DRL algorithms save the data from the interactions in a replay buffer D in the form of transitions (s, a, r, s', d), corresponding to the current state and action, the reward, the next state, and a "done" signal to indicate if the next state is the final state of the environment. When training, we sample batches of random transitions from the replay buffer to update the parameters.

Like any value-based method, we only train a network with parameters ϕ to approximate the action-value function, like the one we show in Figure 5 with discrete actions. Nonetheless, we have a second network, called the target network, that we do not explicitly train, but we use it to train the first one. This target network is a lagged version of the other (i.e. with outdated parameters ϕ_{targ}) that we modify by copying the parameters of the first one every certain number of steps.

The target network helps to avoid the moving target problem when training the other. The loss function is similar to the sub-expression $targets - old\_estimate$ of Equation 4. Thus, we want to make our current estimates $Q(S_t, A_t, \phi)$ more similar to the target $R_t + \gamma max_a Q(S_{t+1}, a, \phi)$ we compute using the same parameters we modify. Consequently, the current estimates cannot reach the target because they are changing too, which makes training unstable. To solve this issue, we use the target network to compute the target, as this second network will not change.

We present the loss function below, called mean-squared Bellman error (MSBE), as it involves using the Bellman equation. It computes an expectation because we sample multiple transitions from the replay buffer, computing the loss with each and averaging the results. Moreover, it uses the "done" signal in the target because if the next state s' is the final one, we cannot execute any action a', so we cannot compute $Q(s', a', \phi_{targ})$. In particular:

$$L(\phi, D) = \mathbb{E}_{(s,a,r,s',d)\sim D}[(Q(s, a, \phi) - (r + \gamma(1 - d)max_{a'}Q(s', a', \phi_{targ})))^2] \tag{14}$$

- **Deep Recurrent Q-Network**:
  Deep Recurrent Q-Network (DRQN) [17] modifies DQN to handle environments where the state does not fulfil the Markov property (i.e. contains information about past transitions), meaning we cannot formulate the problem as an MDP. DQN uses fully connected networks, which do not have memory and can only learn using the current state. Hence, if the state does not contain all the needed information, and we depend on previous states, DQN cannot properly learn the action-value function to solve the environment.

  For this reason, DRQN modifies the architecture of DQN to add an LSTM layer before the output layer to store in memory the information of past states and use it when computing the current action values. The motivation for adding the LSTM comes from how the authors of DQN trained their method on the Atari 2600 games benchmark [2]. Each game is an environment, and we want to learn to solve the game like, or better than, a human would.

  The states are all the game frames (i.e. images), and the actions are discrete. The problem with using a single frame is that there is some information we cannot extract. For example, one frame of the game of Pong will depict the ball's position, but we cannot know its direction or velocity. We would need multiple consecutive frames to extract that information, requiring more inputs than the current state to compute the action values, the original state not fulfilling the Markov property.

  For this reason, the authors of DQN decided to stack four consecutive frames and consider each sequence of four frames the new state. This new state contained all the needed information and fulfilled the Markov property. This solution is valid in this benchmark but not in all environments.

That is the reason the authors of DRQN decided to add the recurrent layer to handle non-Markovian environments.

The training process is very similar to DQN, only changing the way we sample data to train. In DQN, we sample a batch of transitions from the replay buffer to update the parameters. In DRQN, given that we have the LSTM layer, we need to gather complete episodes and feed them sequentially to update the LSTM memory cell. Nevertheless, it minimizes the same loss function and uses the target network to avoid the moving target problem.

- **Deep Deterministic Policy Gradient**:
Deep Deterministic Policy Gradient (DDPG) [23] is the continuous version of DQN. It is an actor-critic off-policy method that learns deterministic policies and only works with continuous actions. Like DQN, it uses target networks to avoid the moving target problem when updating the critic.

  Learning a deterministic policy to output continuous actions has advantages over learning a stochastic one. It is easier to back-propagate through a deterministic function rather than some distribution involving a random factor. Also, after training, it is better to directly output an action rather than sampling from some distribution as we ensure more precise actions with less noise. However, we must add Gaussian noise to the actions to explore the action space enough during training to achieve good policies.

  DDPG does not compute a maximum over all the actions but learns a policy π that directly approximates the maximal action value (i.e. $max_a\, Q(s, a, \phi) \approx Q(s, \pi(s, \theta), \phi)$). Because the action space is continuous, the true action-value function is supposed to be differentiable with respect to the action, allowing it to perform this approximation and avoid the exhaustive maximum, which is very costly when having an infinite number of actions.

  Thanks to that assumption, we can set a gradient expression to train the policy network that only requires moving in the gradient direction that maximizes the action-value function. That is, we compute the gradient of the action-value function and apply gradient ascent. Thus, the actor objective function is as follows:

$$J(\theta, D) = \mathbb{E}_{s\sim D}[Q(s, \pi(s, \theta), \phi)] \tag{15}$$

  To train the action-value network, we compute the MSBE. However, as the actor approximates the action with the highest action value, we do not need to calculate a maximum over all the a's but only evaluate the policy in s'. Moreover, as we have actor and critic networks, we also have their target versions. Thus, when computing $Q(s', a', \phi_{targ})$, we use the target actor to obtain the action (i.e. $a' = \pi(s', \theta_{targ})$). In particular, the critic loss function is as follows:

$$L(\phi, D) = \mathbb{E}_{(s,a,r,s',d)\sim D}[(Q(s, a, \phi) - (r + \gamma(1 - d)Q(s', \pi(s', \theta_{targ}), \phi_{targ})))^2] \tag{16}$$

  Finally, unlike DQN, we do not update the target networks by copying the parameters of the main network every few steps, but we update them each step by applying a polyak average. In the following equation, ρ is a hyperparameter, usually close to 1:

$$\begin{aligned} \phi_{targ} &\leftarrow \rho\phi_{targ} + (1 - \rho)\phi \\ \theta_{targ} &\leftarrow \rho\theta_{targ} + (1 - \rho)\theta \end{aligned} \tag{17}$$

- **Twin Delayed DDPG**:
Twin Delayed DDPG (TD3) [13] is an improved version of DDPG, achieving state-of-the-art results, which adds three main modifications to reduce the overestimation problems of the action-value function approximation [43, 45]. As the name implies, it happens when the approximated value $Q(s, a, \phi)$ surpasses the true value $q_\pi(s, a)$ for a particular state-action pair (s, a).

It happens in all the previous methods we have explained due to the loss function used to update the action-value network. In particular, because of the maximum operation in DQN, or the policy trying to approximate the action with maximum action value in DDPG. The error accumulates within each update and propagates to subsequent state-action pairs, resulting in sub-optimal performance or divergent behaviour because the method tries to exploit these errors (i.e. focuses on wrong actions).

The first modification is maintaining two approximations of the action-value function, using separate networks, and computing the minimum as the target to update both critic networks. With the minimum, we ensure that if one network overestimates the action value for one pair, the other will probably not, and the minimum will cancel the overestimation.

Even though they use the minimum to update both critics, they do not use it to update the single actor and use the first approximation. The authors do not explain why, but we assume this first trick comes from Double DQN [45], a value-based method that maintains two approximations to reduce overestimation but only uses one to compute the actions. Another reason would be trying to back-propagate through the minimum function, which is not completely defined.

In any case, by using only the first critic to update the actor, we can consider this first one the good approximation and the second a backup for when this first one overestimates the action value in some state-action pair, thanks to using the minimum when updating both critics.

The second modification is adding clipped Gaussian noise to the actions coming from the target policy, acting as a regularizer. When computing the actions to explore the environment and gather data to store in the replay buffer, TD3, like DDPG, adds Gaussian noise to those actions. However, TD3 also adds noise to the actions we compute with the target policy when updating the critic networks, but clipped to avoid modifying them too much.

If the critic develops an incorrect peak for a particular state-action pair (due to a big overestimation), the deterministic policy will exploit that action, probably leading to sub-optimal behaviour. By adding this noise, we enforce a generalization such that similar actions will have similar action values, smoothing those values and avoiding those peaks.

The third modification is updating the policy and the target networks at a lower frequency than the critics. Overestimating the action values is easier when the policy is bad, and the policy becomes worse if the action values are inaccurate. If we delay those updates, we minimize the error before performing the policy updates, leading to higher-quality updates.

With these three modifications, the loss equation for both critics, which we want to minimize, is as follows, being $[-c, c]$, the range at which we clip the target Gaussian noise. The Gaussian's σ and c are hyperparameters. In particular:

$$\begin{gathered} L(\phi_i, D) = \mathbb{E}_{(s,a,r,s',d)\sim D}[(Q(s,a,\phi_i) - y(r,s',d))^2] \\ y(r,s',d) = r + \gamma(1-d)(min_{j=1,2}\ Q(s', \pi(s', \theta_{targ}) + \xi, \phi_{targ_j})) \\ \xi \sim clip(N(0,\sigma), -c, c) \end{gathered} \tag{18}$$

The actor objective function, which we want to maximize, is as follows:

$$J(\theta, D) = \mathbb{E}_{s\sim D}[Q(s, \pi(s,\theta), \phi_1)] \tag{19}$$

- **Soft-Actor Critic**:
  Soft Actor-Critic (SAC) [16] is an off-policy actor-critic method that learns stochastic policies for environments with continuous actions, displaying state-of-the-art results. Nonetheless, it is easy to extend to discrete actions [5].

Learning stochastic policies with continuous actions also has some advantages. First, sampling from a distribution naturally leads to exploring the action space without any other mechanism. Second, stochastic policies are more robust to noise or partial observations because we can still sample the optimal action as long as it has some probability.

SAC follows the maximum entropy RL framework, which seeks to maximize the expected return and the policy's entropy (i.e. make all the actions more equally probable). We denote it as $J(\pi)$, as it is how the original article did (do not confuse it with $J(\theta)$, which would be the actor objective function, both seeking to maximize the same). It computes an expectation under trajectories τ ran in the environment, meaning s_t and a_t are not random variables. In particular:

$$\begin{aligned} J(\pi) &= \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) + \alpha H(\pi(\cdot|s_t)] \\ &= \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) - \alpha log\pi(a_t|s_t)] \end{aligned} \tag{20}$$

However, we cannot maximize both terms because they go in opposite directions (i.e. we face a trade-off). At the beginning of training, we want to maximize the entropy to enforce a more exploratory policy. However, when training proceeds and the approximations start to converge, we want to make the policy more deterministic and select the optimal action to maximize the episode return. For this reason, we have the temperature parameter α to balance how much entropy we want, which we can reduce as training proceeds.

Like TD3, SAC maintains two approximations of the action-value function using separate networks, computing the minimum when updating each critic. However, unlike TD3, SAC computes the minimum when updating the actor. Finally, as SAC learns a stochastic policy, the actor outputs the parameters of a Gaussian μ and σ.

However, we do not directly sample the actions from the Gaussian because we cannot back-propagate the gradient through a random process. The gradient tells us how the actor loss changes if we change the policy parameters, but in this case, we will get a different loss without changing the parameters because the action will change each time we sample.

For this reason, the authors apply the reparameterization trick [21] to separate the random process from the gradient computation. In particular, they sample some random noise $\xi \sim N(0, 1)$ from a normal Gaussian and then modify it with the approximated parameters to ensure the modified sample comes from the original Gaussian (i.e. $\mu + \sigma\xi$). With this trick, the gradients flow through the deterministic outputs μ and σ but not through the stochastic one ξ.

With the new objective, the definition of the action-value function also changes to include the entropy term in the expectation. Consequently, the Bellman equation also includes it, and the same with the actor and critic loss functions. In particular, the actor objective function is as follows, which we want to maximize:

$$\begin{gathered} J(\theta, D) = \mathbb{E}_{s\sim D}[min_{j=1,2}\ Q(s, \hat{a}(s, \xi, \theta), \phi_j) - \alpha\, log\, \pi(\hat{a}(s, \xi, \theta)|s, \theta)] \\ \hat{a}(s, \xi, \theta) = tanh(\mu(s, \theta) + \sigma(s, \theta)\xi) \\ \xi \sim N(0, 1) \end{gathered} \tag{21}$$

SAC trains both critics to minimize the MSBE loss, now including the entropy term in the target. Unlike TD3, SAC does not have a target actor, using the only actor to compute the target actions. Moreover, it does not add noise to those actions, as the implicit noise from sampling from

a distribution is enough to smooth the action values. In particular:

$$\begin{gathered} L(\phi_i, D) = \mathbb{E}_{(s,a,r,s',d)\sim D}[(Q(s,a,\phi_i) - y(r,s',d))^2] \\ y(r,s',d) = r + \gamma(1-d)(min_{j=1,2}\ Q(s',\hat{a}',\phi_{targ_j}) - \alpha\, log\, \pi(\hat{a}'|s',\theta)) \\ \hat{a}' \sim \pi(\cdot|s',\theta) \end{gathered} \tag{22}$$

2.6 Multi-Agent Reinforcement Learning

Multi-agent reinforcement learning (MARL) [1] extends RL by having multiple agents interacting with the same environment, each learning some behaviour, and the environment's state changing after each agent executes an action. With MARL, we can model real-world problems that naturally formulate as multi-agent systems, like playing football, a cooperative task with a shared goal. However, we can also model competitive problems where the agents have conflicting goals. We focus on cooperative ones, so we will mainly explain methods addressed to them.

We can still use single-agent methods to solve cooperative multi-agent problems. In that case, we would concatenate the actions of each agent as a joint action and learn a joint policy using the information of the environment's state. The problem is the action space increases exponentially with the number of agents, limiting its application to a few agents or actions per agent. Thus, it makes more sense to use multi-agent techniques.

We cannot model a MARL problem as an MDP, but as a decentralized partially observable MDP (Dec-POMDP) [4], the multi-agent version of partially observable MDP (POMDP) [19]. In MARL problems, the environment's state s is not directly observable (i.e. it is hidden), and the agent can only obtain some partial and noisy representation, which we call observation. For example, the Atari 2600 games were environments we could not formulate as MDPs without the trick of frame stacking but as POMDPs. The state is not observable, but the agents use the observations to decide their actions.

POMDPs consist of the tuple $< S, A, P, R, \Omega, O, \gamma >$. We cannot observe the states from S, but P and R still depend on them. The new elements we identify compared to an MDP are the following:

- Ω is the set of possible observations.
- $O : S \times A \times \Omega \to [0,1]$ is the observation probability function, which defines the probability $O(o|s',a)$ to observe o after taking action a and transitioning to the hidden state s'. In most cases, O is unknown.
- γ: The discount factor we use to compute the discounted return.

Dec-POMDP is the extension to a multi-agent cooperative setting. We call it decentralized because the agents act decentralized (i.e. independently of the other agents) based on their local observations and actions. We define Dec-PODMP with the tuple $< S, U, P, R, \Omega, O, N, \gamma >$. The new or modified elements we identify are the following:

- N is the number of agents.
- U is the set of actions each agent can take. For simplicity, we assume each agent takes the same, but they could not. Each agent a takes action u_a, forming a joint action $\mathbf{u}$. We define the joint action space $\mathbf{U} \equiv U^N$ such that $\mathbf{u} \in \mathbf{U}$.
- $P : S \times S \times \mathbf{U} \to [0,1]$ is the transition probability function defined in terms of the joint action space.
- $R : S \times S \times \mathbf{U} \to \mathbb{R}$ is the reward function defined in terms of the joint action space. In cooperative tasks, we assume a global reward all agents share.

In MARL, a history or trajectory τ is a sequence of consecutive action-observation pairs $\tau \in T \equiv (\Omega \times U)*$. We can also distinguish between the trajectories of each agent and the joint trajectories. Thus, τ_a represents a history of agent a, and $\boldsymbol{\tau} \in \mathbf{T} \equiv T^N$ represents a joint history.

As we deal with partial observability, and the state that fulfils the Markov property is not observable, the policy and the action-value function now depend on a trajectory instead of the hidden state to include information on previous transitions. We define both concepts below, distinguishing between the individual and joint policy:

- $\pi(\mathbf{u}|\boldsymbol{\tau}) : \mathbf{T} \times \mathbf{U} \to [0, 1]$ is the joint policy.
- $\pi_a(u_a|\tau_a) : T \times U \to [0, 1]$ is the policy of agent a.
- $q_\pi(\boldsymbol{\tau}, \mathbf{u}) : \mathbf{T} \times \mathbf{U} \to \mathbb{R}$ is the joint action-value function.

As with single-agent RL, we mostly leverage function approximation to learn the policy and the joint action-value function using neural networks. With all the previous concepts, the goal of a cooperative MARL problem is learning the joint policy, or the individual policies of each agent, that maximizes the joint action-value function at each hidden state.

Independent Q-learning (IQL) [40] is the most naive approach, in which we assume each agent is alone in the environment and the other agents are part of it. We could see it as solving multiple single-agent problems at the same time. While this approach can work in very few cases, it has a lot of flaws. For example, it cannot learn any cooperative behaviour between the agents.

Another important problem is the non-stationarity of the reward and the transition function. To model an environment with an MDP, the transition and reward functions cannot change with time. In MARL, they should depend on the state and the joint action. However, in the previous approach, each agent considers himself alone in the environment, even though the other agents' actions will affect it. Hence, each agent observes how the transition and reward function change independently of its actions, and the agent endlessly tries to adapt to how those functions change in vain.

If the reward function depends on the joint action, each agent needs the other agents' actions to learn its policy and action-value function. One option is for the agents to share or communicate information. However, communication is not usually available. The most common method is to predict the actions of the other agents. Nonetheless, we need their information to predict those actions, and each agent has several actions, scaling exponentially (i.e. curse of dimensionality). To make matters worse, each agent is learning, so each agent needs to adapt to the policy of the others changing too, creating a problem of moving target.

IQL's opposite approach would be centralizing the training and the execution, having a centralized component to control all the agents. Thus, assume a single agent and leverage single-agent techniques. Hence, the best option is a compromise between both approaches. The most common approach is leveraging the framework of centralized training with decentralized execution (CTDE) [29], which helps solve the problems of non-stationarity and the moving target. Moreover, it also helps in learning coordinated behaviours with the constraints of partial observability and limited or no communication.

CTDE trains the networks assuming the ideal situation of available communication and full-observability. Hence, it has a centralized component with access to the hidden state and the agents' actions and observations. Using that information, CTDE guides the learning of the agents' policies. During execution, we cannot make these assumptions, so we do not use the centralized component, and each agent acts based only on its observations and actions. Hence, we learn decentralized policies in a centralized fashion.

For example, in an actor-critic setting, the critic is centralized to use all the available information, and the actors are decentralized and use their local information to make decisions. The critic is used only during training, but the actors also after training to interact with the environment.

When training the decentralized agents, a technique usually used is to train a single network instead of one per agent, reducing the number of parameters to train. One might think that sharing weights will generate a single policy, and all agents will behave identically. However, we can achieve different behaviours by adding an agent identifier (i.e. one-hot encoding) as an extra input [34, 38, 12]. The network learns to pay attention to the identifier to know the type of behaviour it needs to output for each agent.

Training one network has the limitation that each agent needs the same observation and action space, or at least share dimensions, as the network will have a fixed input and output shape. We can use padding methods like the one in [42] when they have different dimensions. Nevertheless, this approach of sharing weights does not work in most cases, and if the agents do not behave similarly, we need to train separated networks.

One important challenge CTDE does not solve is the multi-agent credit assignment problem [54, 9]. In cooperative tasks, we have a global joint reward function that gives a reward based on the agents' coordinated actions. However, we cannot know the contribution of each agent to the joint reward. Ideally, we would like a reward function for each agent, but in most cases, we do not have one. And if we have, they often fail to encourage the individual agent to sacrifice for the greater good, which results in failing to learn challenging tasks.

3 State of the Art

We will briefly explain in this section different methods to solve initially single-agent environments with a very large action space. Our solution is decomposing the actions into multiple agents and leveraging MARL techniques. However, other approaches maintain a single agent and perform other tricks to handle those action spaces. In subsection 3.1, we will present methods that keep the single-agent setting, and in subsection 2.6, we will present different MARL approaches.

Nonetheless, we want to mention that when explaining each approach, we will follow its article notation as much as possible but also try to maintain a similar one to the one used in section 2. For this reason, even though one sees the action-value function represented as Q but does not see the network parameters ϕ, that does not mean we are not referring to the approximation the networks learn. The authors decided not to include it.

3.1 Action Space Decomposition

When searching for the literature on single-agent methods to deal with enormous action spaces (e.g. millions of actions), most work with environments with discrete actions. And if they experiment with continuous ones, their solution is to discretize the action space. Besides, in most cases, they implement extensions or modifications to DQN to deal with the computation of the maximum over all the actions when evaluating and updating the policy.

It may seem more difficult to deal with large action spaces only with value-based methods like DQN. However, with actor-critic methods, which we usually use with continuous actions, even though we do not need to evaluate the whole action space, we still need to explore enough actions to learn the policy and the value function. Hence, decomposing a continuous action space to avoid having a single network learning to generalize through millions of actions should also help, not only with value-based methods

and discrete actions.

The first method is Branching Dueling Q-Network (BDQ) [41], a branching variant of Dueling Double DQN [49], which also approximates the action-value function by aggregating the advantage and the state-value function. The advantage function $A(s,a)$ [49, 37], defined as $A(s,a) = Q(s,a) - V(s)$, tells if taking action a in state s is better than the expected action in s.

BDQ assumes a discrete N-dimensional action space, each dimension d having n sub-actions. The method trains a network receiving the state s as input. First, some layers compute an embedded representation of s, later feed to $N+1$ sub-network branches. One approximates the state-value function, and the others, the advantage $A_d(s,a_d)$ of all the sub-actions $a_d \in A_d$ in dimension d, one branch per dimension. Then, each sub-action branch merges the advantages of each sub-action with the state value $V(s)$ to compute the action values $Q_d(s,a_d)$ using the following aggregation layer:

$$Q_d(s,a_d) = V(s) + (A_d(s,a_d) - \frac{1}{n} \sum_{a'_d \in \mathcal{A}_d} A_d(s,a'_d)) \tag{23}$$

Once each sub-action branch computes all the action values, it computes an *argmax* to obtain the greedy sub-action, later merged to obtain the greedy action. The advantage of BDQ over DQN is that instead of having n^N outputs to evaluate to extract the greedy action, we have N *argmax* of n sub-actions each. BDQ updates the network by computing the MSBE per each sub-action branch and averaging the results.

The network uses the shared state representation because the authors believe that back-propagating the gradient of all the branches to this shared state representation generates a richer set of features that help coordinate the different sub-action dimensions, which in turn helps improve training stability and convergence.

A second approach is Amortized Q-learning (AQL) [50], a simple method to scale up DQN to large action spaces, either discrete or continuous. It tackles the problem of DQN by learning a distribution μ over actions using supervised learning and then sampling a reduced set from that distribution to compute the maximum.

AQL works with both action types because it leverages explicit search and supervised learning to approximate the action with the highest Q-value instead of making DDPG assumptions to update the policy parameters by maximizing the action values. Thus, not computing gradients with respect to the actions makes AQL agnostic to the action type.

AQL trains one network to approximate the action-value function and another to compute μ given the state s. AQL replaces the maximum over all the actions with a maximum over a reduced set. This reduced set comes from the union of two: M actions from the uniform distribution and N from $\mu(\cdot|s)$. Both sizes are hyperparameters, so we want to learn a distribution from which we draw the fewest number.

We denote with $a^*(s)$ the action with maximum action value from the union of the two sets. We use it in the action-value network loss when computing the target estimate and in ϵ-greedy action selection to denote the best action. We also use it to train the network to learn μ in a supervised way to minimize the following loss:

$$L(\theta^\mu, s) = -log\mu(a^*(s)|s,\theta^\mu) - \lambda H(\mu(a|s,\theta^\mu)) \tag{24}$$

The first term is the negative log-likelihood with the maximal action as the target. Hence, minimizing it makes the sample with the highest action value more probable under the distribution μ. The second term, controlled by the hyperparameter λ, is the negative entropy of the distribution to encour-

age uncertainty and avoid the distribution from becoming deterministic and potentially becoming stuck in a bad local optimum.

By changing the form of μ, we can apply AQL to discrete or continuous action spaces. For the former, we use a softmax to generate a categorical distribution, and for the latter, we learn the mean of a Gaussian distribution with fixed variance.

A third approach is [8], which presents an actor-critic architecture called Wolpertinger architecture to deal with the problem of DQN when the action space is huge. Nonetheless, the method only works with discrete actions. According to the authors, dealing with such environments requires sub-linear complexity relative to the action space and the ability to generalize over actions. Value-based methods like DQN can generalize, even to previously unseen ones, but have linear complexity. Actor-critic methods have sub-linear complexity but do not generalize very well.

However, leveraging prior information about the actions to embed them in a continuous space should allow the actor network to generalize to even unseen actions, achieving both requirements. Moreover, the embedding decouples the policy's complexity from the action space's cardinality. For this reason, the authors use DDPG to train both the actor and the critic, even though they only consider discrete actions.

The actor outputs continuous action, called proto-action. Thus, to map them to those in the original discrete space, the architecture computes an approximated k-nearest neighbour (K-NN) [28] with logarithmic complexity to find the closest ones to the proto-action. Instead of simply keeping the closest one, which is a naive approach as we can have outliers which are closer but have a low action value, it is better to use the k closest ones and choose the one with the highest action value, obtained by evaluating the critic.

The method trains both actor and critic using DDPG, meaning that the losses are the same as in DDPG. The actor loss uses the continuous proto-actions, while the critic uses the discrete ones obtained after performing the K-NN and the *argmax*, later stored in the replay buffer. Training the critic using discrete actions avoids ignoring the information about the real actions executed in the environment.

A fourth approach is Action-Elimination DQN (AE-DQN) [53], which combines DQN with the concept of action elimination. Action elimination keeps, in each state, only the most optimal actions, avoiding exploring the irrelevant ones. It also helps to converge to better policies because we only need to take the maximum on a subset, reducing overestimation errors. Moreover, the action value approximations must be accurate only for these valid actions. Thus, we need to sample fewer actions for the action-value function to converge, and we can learn a simpler function approximation.

AE-DQN assumes the environment returns at each time step a binary elimination signal $e(s, a)$ that gives feedback about the optimality of actions. 1 means a might be eliminated in s, and 0 otherwise. This feedback is immediate, unlike the reward that depends on the long-term consequences. Thus, learning the actions to eliminate is faster than learning the optimal actions using only the reward.

It uses this elimination signal to determine the subset of actions to keep. However, the method does not use it as it comes, but AE-DQN learns to approximate it using an extra network called Action Elimination Network (AEN). We will not enter into details about how to approximate the signal. However, the authors argue that while the signal is binary, we can relax the assumption and allow the expectation to take values in the $[0, 1]$ range. To train this AEN, it minimizes the MSE loss using the original binary elimination signal as the label.

With the approximated elimination signal, when we need to evaluate the action with maximum action value, we compute the elimination signal for all the actions in state S_t and keep those with a

signal lower or equal than a hyperparameter l (i.e. $l \approx 0.5$). Within that reduced set, we compute the maximum and find the best action.

The method uses the AEN to decouple the action elimination from the action-value approximation, allowing it to leverage the standard DQN. Hence, while the AEN eliminates irrelevant actions to balance exploration and exploitation, DQN explores and learns action values for valid actions, not those eliminated.

3.2 Multi-Agent Reinforcement Learning

We can distinguish two types of methods following the CTDE paradigm: those factoring the joint action-value function approximation, which we refer to as Q_{tot}, and those that do not. Factoring it consists of computing it as a function of some utility functions to represent each agent's contribution to the joint reward, which we refer to as $[Q_a]_{a=1}^N$, such that computing the *argmax* of Q_{tot} leads to the same result as concatenating the *argmax* of each Q_a.

We consider $[Q_a]_{a=1}^N$ utility functions because they do not represent a real action-value function. We parameterize each Q_a by the agent's a observations and actions. Thus, even though the environment could have a reward per agent instead of a shared one, that reward would still be conditioned by the hidden state and the joint action. Hence, the real individual action-value function approximation would receive the hidden state (or the agents' observations) and all agents' actions as input.

We call this equivalence Individual-Global-Max (IGM) [38], and we can formally define it as follows. For a joint action-value function $Q_{tot} : \mathbf{T} \times \mathbf{U} \rightarrow \mathbb{R}$, if there exist individual action-value functions $[Q_a]_{a=1}^N$ such that the following equivalence holds, then $Q_{tot}(\boldsymbol{\tau}, \mathbf{u})$ is factorized by $[Q_a(\tau_a, u_a)]_{a=1}^N$.

$$argmax_{\mathbf{u}} Q_{tot}(\boldsymbol{\tau}, \mathbf{u}) = \begin{pmatrix} argmax_{u_1} Q_1(\tau_1, u_1) \\ \vdots \\ argmax_{u_N} Q_N(\tau_N, u_N) \end{pmatrix} \tag{25}$$

When training, we have one or multiple networks to compute $[Q_a]_{a=1}^N$ and an extra network (if needed) that mixes them into Q_{tot}. To compute the loss, we use Q_{tot} because we have a single shared reward, and then we back-propagate through the networks computing each Q_a.

Factoring the joint action-value function has some advantages. We do not need a centralized component that receives the hidden state (if available) and all the agents' observations and actions to learn Q_{tot}. We only need to mix the individual action-value functions $[Q_a]_{a=1}^N$, which is easier and scales better in large environments. Besides, retraining the mixing network if we increase the number of agents is faster than learning that monolithic component.

Another advantage is that methods factorizing Q_{tot} handle the credit assignment problem implicitly as they learn to decompose Q_{tot} (i.e. the accumulated global reward) into the individual action-value functions (i.e. each agent contribution to that joint reward) and use those individual functions to update the policy or select actions from them.

In comparison, methods that do not factorize Q_{tot} need to use some explicit mechanism to learn the agents' contribution or use the joint reward and still try to learn optimal policies for each. The latter type is the most difficult, as the network needs to extract from a single scalar value the contributions of all agents. Maybe one agent executed a sub-optimal action, but the joint reward was high, and the policy would increase its probability.

Different methods factorize the joint action-value function differently, all fulfilling the IGM principle.

Two of the most known, both value-based, are value-decomposition networks (VDN) [10] and QMIX [34]. The first factorizes Q_{tot} as a sum, and the second computes a non-linear function using a feed-forward network. To fulfil IGM, QMIX ensures a monotonic constraint on the relationship between Q_{tot} and each Q_a by forcing the mixing network to have positive weights learned using hypernetworks [15]. In particular:

$$\text{VDN:} \quad Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}) = \sum_{a=1}^{N} Q_i(\tau_a, u_a) \tag{26}$$

$$\text{QMIX:} \quad \frac{\partial Q_{tot}(\boldsymbol{\tau}, \mathbf{u}, s)}{\partial Q_a(\tau_a, u_a)} \geq 0, \ \forall a \in N \tag{27}$$

With QMIX, we can learn to approximate any joint action-value function that we can factor into a non-linear monotonic combination of $[Q_a]_{a=1}^N$. While both VDN and QMIX sufficiently meet the IGM principle, they can be excessively restrictive, especially in tasks where the joint action-value function does not meet the additive and monotonicity constraints. For this reason, other methods were designed to cover a wider class of MARL tasks.

One example is QTRAN [38], another value-based method which does not present any structural constraint when factorizing the joint action-value function and aims at factorizing any task. The key idea of QTRAN's factorization is to transform the original Q_{tot} into a new Q'_{tot} that shares the same optimal joint action and is easier to factorize. To do so, it learns a state-value function V_{tot} to correct the discrepancy between Q_{tot} and Q'_{tot}, which usually arises from the partial observability of agents.

The authors present a theorem stating the sufficient condition for $[Q_a]_{a=1}^N$ to satisfy the IGM principle, thus Q_{tot} being factorized by $[Q_a]_{a=1}^N$. Being $\hat{u}_a = argmax_{u_a} Q_a(\tau_a, u_a)$ the optimal local action, and $\hat{\boldsymbol{u}} = [\hat{u}_a]_{a=1}^N$ the optimal joint action, we have:

$$\begin{gathered} \sum_{a=1}^{N} Q_a(\tau_a, u_a) - Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}) + V_{tot}(\boldsymbol{\tau}) = \begin{cases} 0 & \boldsymbol{u} = \hat{\boldsymbol{u}} \\ \geq 0 & \boldsymbol{u} \neq \hat{\boldsymbol{u}} \end{cases} \\ V_{tot}(\boldsymbol{\tau}) = max_{\boldsymbol{u}} Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}) - \sum_{a=1}^{N} Q_a(\tau_a, \hat{u}_a) \end{gathered} \tag{28}$$

QTRAN defines Q'_{tot} as a linear sum of $[Q_a]_{a=1}^N$, so we know it is a valid factorization according to IGM. Then, it learns $[Q_a]_{a=1}^N$, Q_{tot}, and V_{tot}, using three networks and a composite loss with three terms. The first term is the loss for estimating Q_{tot} (i.e. the MSBE). The second ensures the optimal action $\hat{\boldsymbol{u}}$ satisfies the first case of the sufficient condition (i.e. $= 0$). The third term ensures the non-optimal action $\boldsymbol{u}$ satisfies the second case (i.e. ≥ 0). Thus, we learn to factorize Q_{tot} from a simpler factorization by optimizing the previous loss.

The authors of QTRAN demonstrated how their method achieved better results than VDN and QMIX in tasks where the joint action-value function did not meet the additivity or monotonicity constraints. Even though QTRAN does not impose any constraint in the factorization and should factorize any function, it relaxes the IGM consistency between the joint and individual greedy action selection (i.e. Equation 25), causing QTRAN to suffer from instability and perform badly in complex environments.

QPLEX [47] is a value-based method that appeared to overcome the problems of the previous methods and achieve complete expressiveness of the IGM function class. It leverages a duelling architecture [49], like BDQ, to learn to approximate the state-value and the advantage functions and then aggregate both to compute the action-value function (i.e. $Q = V + A$). However, we distinguish between V_a and

V_{tot}, and A_a and A_{tot}, computing Q_{tot} and each Q_a using the duelling architecture as follows:

$$Q_{tot}(\boldsymbol{\tau}, \mathbf{u}) = V_{tot}(\boldsymbol{\tau}) + A_{tot}(\boldsymbol{\tau}, \mathbf{u}) \ \textit{where} \ V_{tot}(\boldsymbol{\tau}) = max_{\mathbf{u}'} Q_{tot}(\boldsymbol{\tau}, \mathbf{u}') \tag{29}$$

$$Q_a(\tau_a, u_a) = V_a(\tau_a) + A_a(\tau_a, u_a) \ \textit{where} \ V_a(\tau_a) = max_{u'_a} Q_a(\tau_a, u'_a) \tag{30}$$

From the perspective of the duelling architecture, the IGM consistency should only constrain the action-dependant advantage and be free of the state-value function, motivating QPLEX authors to reformulate IGM as advantage-based IGM and transform the consistency constraint onto advantage functions. We can define the consistency constraint of advantage-based IGM as follows:

$$argmax_{\mathbf{u}} A_{tot}(\boldsymbol{\tau}, \mathbf{u}) = (argmax_{u_1} A_1(\tau_1, u_1), \dots argmax_{u_N} A_N(\tau_N, u_N)) \tag{31}$$

One advantage of advantage-based IGM is that we can achieve this constraint by limiting the value range of the advantage functions. That is, the previous *argmax* constraint is equivalent to the following one in which we are restricting the values of the joint and individual advantage functions for both optimal actions $\mathbf{u}^*$ and u_a^*, and non-optimal actions $\mathbf{u}$ and u_a:

$$\begin{aligned} A_{tot}(\boldsymbol{\tau}, \mathbf{u}^*) = A_a(\tau_a, u_a^*) = 0 \\ A_{tot}(\boldsymbol{\tau}, \mathbf{u}) < 0 \\ A_i(\tau_a, u_a) \leq 0 \end{aligned} \tag{32}$$

With this equivalent expression, the authors developed an efficient MARL algorithm that learns the joint state-value function with any scalable decomposition structure and just imposes simple constraints limiting the value ranges of the advantage functions.

QPLEX first learns each Q_a using a recurrent action-value network, and from that, it computes each V_a and A_a instead of the other way around one usually does with the duelling architecture. In particular, it computes V_a as a maximum over the actions (i.e. $V_a(\tau_a) = max_{u_a} Q_a(\tau_a, u_a)$), and substracts V_a from Q_a to compute A_a (i.e. $A_a(\tau_a, u_a) = Q_a(\tau_a, u_a) - V_a(\tau_a)$).

Then, it computes the joint action-value function with a centralized component with two modules. The first one is a transformation network that incorporates during training centralized information into the individual V_a and A_a with a positive linear function that maintains the consistency constraint:

$$\begin{aligned} V_a(\boldsymbol{\tau}) = w_a(\boldsymbol{\tau}) V_a(\tau_a) + b_a(\boldsymbol{\tau}) \\ A_a(\boldsymbol{\tau}, u_a) = Q_a(\boldsymbol{\tau}, u_a) - V_a(\boldsymbol{\tau}) = w_a(\boldsymbol{\tau}) A_a(\boldsymbol{\tau}, u_a) \end{aligned} \tag{33}$$

The second is a duelling mixing network that composes the outputs from the transformation network into a joint action-value function:

$$\begin{aligned} Q_{tot}(\boldsymbol{\tau}, \mathbf{u}) = V_{tot}(\boldsymbol{\tau}) + A_{tot}(\boldsymbol{\tau}, \mathbf{u}) \\ V_{tot}(\boldsymbol{\tau}) = \sum_{i=a}^{N} V_a(\boldsymbol{\tau}) \\ A_a(\boldsymbol{\tau}, \mathbf{u}) = \sum_{a=1}^{N} \lambda_a(\boldsymbol{\tau}, \mathbf{u}) A_a(\boldsymbol{\tau}, u_a) \end{aligned} \tag{34}$$

It computes V_{tot} as a summation because advantage-based IGM does not impose any constraint on state-value functions. However, to compute A_{tot} to enforce the consistency, it calculates a dot product using a set of positive weights $\lambda_a(\tau, \mathbf{u})$ learned with multi-head attention [46]. Similar to QTRAN, A_{tot} can be seen as a term to correct the discrepancies between V_{tot} (i.e. a simpler factorization) and Q_{tot}. However, thanks to using self-attention, the joint information of λ_a provides the full expressiveness power of value factorization to overcome the limitations of previous methods.

Residual Q-Networks (RQN) [32] is a value-based method that achieves state-of-the-art results like the previous methods. However, RQN converges faster, with increased stability and robust performance. Moreover, it does not take advantage of the hidden state information during training, allowing it to apply to non-simulated environments.

RQN applies the same idea of QTRAN of transforming Q_{tot} into a new function with the same optimal action that factorizes using a summation. However, RQN corrects the discrepancies by learning, through a residual network, individual correction factors $[\phi_a(\boldsymbol{\tau})]_{a=1}^N$ that correct $[Q_a]_{a=1}^N$, instead of correcting the discrepancies with the joint function using V_{tot} as QTRAN does. Thus, RQN gives more flexibility to learn the individual maximum action values. We can re-formulate Equation 28 using these correction factors:

$$\sum_{a=1}^{N}(Q_a(\tau_a, u_a) + \phi_a(\boldsymbol{\tau})) - Q_{tot}(\boldsymbol{\tau}, \mathbf{u}) = \begin{cases} 0 & \mathbf{u} = \hat{\mathbf{u}} \\ \geq 0 & \mathbf{u} \neq \hat{\mathbf{u}} \end{cases} \tag{35}$$

Like other methods, RQN trains individual networks to approximate Q_a for each agent a, using batches of episodes. Then, these action values enter a residual network that computes the adjusted action values $Q_a^+ = Q_a + \phi_a$ by learning the correction factors ϕ_a. The complete path of the network computes ϕ_a, and a skip connection brings Q_a to the output to compute Q_a^+. With the adjusted values, it computes Q_{tot} as $Q_{tot}^+ = \sum_{a=1}^N Q_a^+(\tau_a, u_a)$.

However, the residual network uses as input features the mean and maximum action value over the steps of each episode in the mini-batch. The intuition for using the maximum is that if the trajectory of an agent contains some high-valued state-action pair, the agent should be more encouraged to visit it in subsequent episodes. Similarly, the mean value tells if the sequence of state-action pairs is worth visiting again. Hence, these correction factors indicate the relative importance of each trajectory.

RQN solves more tasks than previous methods thanks to the residual connection when computing Q_a^+. In the situation where $Q_a^+ = Q_a$ and no correction is needed, learning this approximation (i.e. the identity function) with another type of network would be difficult. However, the residual Q-network generates a factor ϕ_a that will converge to a stable value when reaching the training objective, becoming obsolete, and Q_a taking the shortcut.

Counterfactual Multi-Agent Policy Gradients (COMA) [9] is an actor-critic method not decomposing the joint reward but addressing the problem of multi-agent credit assignment with an explicit mechanism. It uses the idea of difference rewards [52], in which each agent a updates its policy using a shaped reward that compares the global reward to the reward received if that agent would have taken a default action:

$$D_a = R(s, \mathbf{u}) - R(s, (\mathbf{u}_{-a}, c_a)) \tag{36}$$

If any action u_a improves D_a, it also improves the global reward $R(s, \mathbf{u})$, as $R(s, (\mathbf{u}_{-a}, c_a))$ ignores that agent's action and considers the default action c_a. The problem is that we need to estimate $R(s, (\mathbf{u}_{-a}, c_a))$ (via simulation or function approximation) and choose the default action c_a for each agent. For this reason, COMA avoids these problems by using a centralized (but not factorized) critic to estimate $Q_{tot}(s, \mathbf{u})$ (assuming s available or $\boldsymbol{\tau}$ if not) to compute the difference rewards using the advantage function. In particular, for each agent a, it computes the following joint advantage $A_{tot_a}(s, \mathbf{u})$:

$$A_{tot_a}(s, \mathbf{u}) = Q_{tot}(s, \mathbf{u}) - \sum_{u'_a} \pi_a(u'_a|\tau_a) Q_{tot}(s, (u_{-a}, u'_a)) \tag{37}$$

The advantage function determines how good an action is in a particular state compared to the expected action. The second operand in the above expression is precisely the expectation over the actions the agent a can take. Thus, we still seek the action that improves the difference between the true reward

and the counterfactual (i.e. the one replacing the action of agent a), and we use it to learn the policies.

Using the advantage does not require any extra simulation or the default action, and we can learn the action-value function using stored experiences in a replay buffer. Nevertheless, computing the advantages is still expensive. COMA does it in a single forward pass of the actor and critic for each agent, thanks to the critic architecture. Each critic a receives the other agents' actions as extra input to compute Q_{tot} for all agent a actions. Then, it uses the action sampled from the policy to calculate A_{tot_a}. Even though the critic has a large input space that scales linearly, it has a linear number of outputs instead of exponential.

Multi-agent soft actor-critic (mSAC) [33] is an actor-critic method factorizing Q_{tot}, and the multi-agent version of SAC. mSAC shares the same loss functions (now using Q_{tot}) and tricks when training the actor, critic, and mixer network, like maximizing the entropy and using two critic and mixer estimations to reduce overestimation problems. It factorizes Q_{tot} linearly and monotonic, as a weighted sum of the individual Q_a, learning the weights $k^a \geq 0$ and b with separate hyper-networks that use the state as input:

$$Q_{tot}(\boldsymbol{\tau}, \mathbf{u}) = \sum_a k^a(s)\mathbb{E}_{\pi_a}[Q_a(\tau_a, u_a)] + b(s) \tag{38}$$

The authors took this factorization from [48], which points out some challenges of multi-agent policy gradient methods that cause inferior performance compared to value-based ones. The authors designed the previous factorization to tackle those challenges and achieve comparable or higher performance. Even though a linear factorization may have limited expressivity, which may introduce bias in value estimations, the authors demonstrated that this limitation does not violate the policy improvement guarantee of the method.

Even though SAC works with continuous actions, the authors of mSAC considered only discrete settings. They demonstrated that mSAC outperformed COMA and achieved competitive results with QMIX. They also presented a version (mCSAC) that adopts the insights of COMA and uses a counterfactual advantage function to deal with the multi-agent credit assignment using COMA's actor loss. Between mSAC and mCSAC, they achieved similar performance in simpler tasks. However, in more complex ones, mCSAC achieved better results.

Learning Implicit Credit Assignment (LICA) [54] is an actor-critic method that does not decompose Q_{tot} into the different Q_a but still learns implicit credit assignment without any mechanism like COMA, using a monolithic and centralized critic formulated similar to the mixer in QMIX. In the environments the authors experimented with, LICA achieved better results than some previous methods. The authors argue that credit assignment does not require an explicit formulation like COMA as long as we fulfil two conditions:

- The policy gradients computed using the centralized critic carry sufficient information for the decentralized agents to maximize their joint action value through optimal coordination.
- We enforce a sustained level of exploration during training.

The key motivation of LICA is that if one can optimize the decentralized policies to maximize the joint action-value function, then any converged policy should have acquired optimal cooperative behaviour without any explicit formulation of credit assignment. However, this insight can be unrealistic and unscalable in complex scenarios. In practice, if a trained critic provides a good approximation, a policy optimization setting that improves along Q_{tot}'s gradient should act as a proxy to find optimal credit assignment strategies.

To this end, LICA formulates a centralized critic that computes Q_{tot} from the joint action and true environment state. Similar to the mixer in QMIX, it uses the state as input to different hypernetworks

to generate the weights and biases of two linear layers that map all agents' actions into the joint action value. The authors use this formulation because unlike an MLP, it integrates an extra representation of the state into the Q_{tot} gradients. Hence, it provides more information about the environment for the agents to address credit assignment implicitly through learning.

LICA is a policy-based method and, like others, shares the limitation that insufficient exploration may lead to premature convergence to poor policies. Like SAC, most rely on an entropy regularization term in the policy loss to encourage exploration, weighted by some constant β.

$$\mathcal{H}(\pi_a(\cdot|\tau_a)) = \beta H(\pi_a(\cdot|\tau_a)) = \beta \mathbb{E}_{u_a \sim \pi_a}[-log\pi_a(u_a|\tau_a)] \tag{39}$$

We want to penalize a low policy entropy resulting from overconfident actions. However, this is not always the case as β has high sensitivity in complex environments, and a small change can lead to drastically different trajectories. Moreover, once a policy starts to converge, the same regularization term may not encourage further exploration.

The authors hypothesize that these issues arise from the derivative of the entropy function, which in turn influences the magnitudes of the policy gradients on different action probabilities. If we want to update the policy of an agent a, which has k discrete actions, we define $p_a = \pi_a(\cdot|\tau_a)$ as the vector of probabilities. Then, the derivative of $\mathcal{H}$ with respect to p_a is:

$$\partial\mathcal{H} = [\frac{\partial\mathcal{H}}{\partial p_a^1}, \ldots, \frac{\partial\mathcal{H}}{\partial p_a^k}] = [-\beta(logp_a^1 + 1), \ldots, -\beta(logp_a^k + 1)] \tag{40}$$

Since each $\partial\mathcal{H}_i = -\beta(logp_a^i + 1)$ is log-shaped, the gradient magnitudes for dampening high-confidence actions (i.e. with high probability) are disproportionately small compared to those encouraging low-confidence actions. Even though favouring the latter should help achieve the former, in practice, insufficient penalties for over-confidence can still result in agents sticking to a subset of high-confidence actions.

For this reason, the authors propose to control the magnitudes of the entropy gradients such that they are inversely proportional to the entropy itself. With this technique, the regularization strength is now adaptive based on the policy stochasticity and a constant ξ instead of depending on β:

$$\partial\mathcal{H}_i = -\xi\frac{logp_a^i + 1}{H(p_a)} \tag{41}$$

TransfQMIX [14] is a value-based method based on QMIX that uses transformers to learn the individual agent networks and the mixer. The authors argue that previous methods mostly focus on how to factorize Q_{tot}, forgetting about the structure of the observation and state space and the architecture of the agents and mixer networks. It greatly outperformed methods like QMIX, QTRAN or QPLEX in all SMAC scenarios the authors experimented with.

TransfQMIX generalizes a set of z features (e.g. the position and velocity) defined for the different entities in the environment, not only the agents. It redefines the observation as the value of those z features for the different k entities the agent observes, resulting in an observation matrix with dimension $k \times z$. It does the same for the state, even though the features may change because we can assume full observability.

It uses transformers [46] to build the individual agents' network and the mixer because we can see the observation features as vertices of a coordination graph, learning the edges through self-attention. By learning this graph, the agents and the mixer can reason about the coordination of each agent and the team. Consequently, TransfQMIX uses transformers to leverage the graph structure and learn better coordination policies.

The input to the transformers is not the entity matrices but an embedding computed using a shared embedding matrix. The advantage of using such a matrix is that it processes each feature type with the same weights, making the matrix independent of the number of entities k and making the approach more scalable and transferable than RNNs, which would depend on k.

The transformer agent receives the embedded observation matrix and a hidden vector initialized with zeros as input. Multi-head self-attention refines the graph vertices based on the attention given to the others and the hidden vector (i.e. it learns the edges of the coordination graph). Like BERT [7] does with the [CLS] token, the transformed hidden vector encodes the coordination reasoning of that agent, computing from it the action values for all actions, sampling one using ϵ-greedy exploration.

The transformer mixer is a two-layer MLP to project the individual Q_a over Q_{tot} like QMIX but generates the weights using the transformer instead of hypernetworks. The transformer receives the agents' hidden vectors, the embedded state matrix, and three recurrent vectors as input. It refines the hidden vectors into W_1 and the three recurrent vectors into b_1, W_2, and b_2, making W_1 and W_2 positive to ensure monotonicity.

The hidden vectors are the coordination reasoning of each agent that we enhance with the other agents' information and the true state of the environment. Each agent a generated its Q_a from those same hidden vectors, later multiplied by W_1. Hence, the mixer combines and refines the independent agents' reasoning so that each agent's refined hidden vector represents the team coordination.

All the previous algorithms work, or at least have been evaluated, with environments having discrete actions. However, one of our focuses is methods that can solve those having continuous action spaces without discretizing the actions first. Like with action decomposition, there is less research on these methods, but we will present some we found that achieve decent results or seem promising.

The first two are the continuous versions of QMIX and VDN, named COMIX and COVDN [22]. The authors developed them as a baseline to compare another method, but we still present them as the idea is simple. With continuous actions, we cannot compute the *argmax* when exploring and computing the targets to update the action-value network. The solution they found was to use the cross-entropy method [20, 22], an iterative method that refines a Gaussian distribution from which it samples the actions.

Starting from the standard normal distribution, it draws a set of N samples at each iteration, keeping the best M (i.e. the ones with the highest action value). With these best samples, it computes their mean and standard deviation to use them as the Gaussian parameters in the next iteration. The process repeats until, at the last iteration, it gathers the best actions and returns them.

Another method is MADDPG [24], the multi-agent version of DDPG. It trains a centralized and monolithic critic per agent, receiving all agents' observations and actions. Thanks to having multiple critics, MADDPG can solve competitive tasks where each agent has a reward function. However, in cooperative ones, where we have a shared reward, all critics learn to approximate Q_{tot} using the same information, not taking advantage of training one per agent.

Thus, MADDPG does not factorize Q_{tot} and does not apply any explicit credit assignment mechanism like COMA. As we explained in LICA, we should still be able to learn coordinated behaviours. However, the same LICA authors argued that MADDPG's critic formulation, an MLP, is too simple, resulting in MADDPG not learning a proper credit assignment and usually failing to learn well-coordinated policies.

Moreover, each agent optimizes its policy by assuming all other agents' actions are fixed. Thus, when computing the actor loss, MADDPG samples them from the replay buffer. However, the other agents' policies may have changed drastically. Optimizing this way can lead to converging to sub-optimal policies because it considers the effect of potentially random actions from the other agents.

FACMAC [31] solves some of the problems of MADDPG by combining it with QMIX. Like MADDPG, It learns a deterministic joint policy. However, instead of training a monolithic critic per agent to approximate Q_{tot}, it learns a centralized but factored critic. In particular, it factorizes Q_{tot} into $[Q_a]_{a=1}^{N}$ by leveraging some critics to learn the individual Q_a and a mixer network to merge them into Q_{tot}, using the same mixer as QMIX if we learn a monotonic factorization.

Compared to MADDPG, factorizing the joint function gives the method better scalability to more agents, as the critics do not receive all the agents' observations and actions. Moreover, using an actor-critic approach allows a more flexible factorization thanks to the unconstrained representation of the critic, for example, non-monotonic, which QMIX could not.

FACMAC trains the policies together instead of computing a different policy gradient for each agent. Updating them together leads to sampling all the actions from their respective policies instead of some from the replay buffer. Thus, it solves the problems of sub-optimal performance MADDPG had. Training the policies together takes complete advantage of a centralized critic because, as [25] remarked, merely using it with per-agent gradients does not necessarily lead to better coordination.

4 Environments

We will present the different environments we have used in our experiments. As we have explained in subsection 1.1, we wanted to search for single-agent environments, or at least environments we could formulate as single-agent, with very big action spaces we could decompose into multiple agents, and leverage MARL techniques.

To decompose the environment into multiple agents, we had to fulfil some conditions we explained in subsection 1.1. For example, we need to divide the action space among the agents, so we have selected environments where the actions are multi-dimensional vectors, assigning one or more dimensions to each agent.

Moreover, we wanted to search for environments we could make more complex by increasing the action dimensionality and, if possible, the agents behaving identically. In that case, we could learn a single behaviour, increase the number of agents, and reuse it with all the agents, including the new ones. All but one of our environments fulfil this requirement, the last one, which we use with another major purpose.

Finally, we wanted to focus this thesis on environments having continuous actions, as most research focuses on discrete ones, considering what we have seen in section 3. All the environments we have used in our experimentation have continuous actions, except one accepting both action types.

In particular, we have chosen three different environments, even though we have used the same one in several experiments. For example, we have used the one accepting the two action types two times, one with each, to demonstrate how we can speed up learning by leveraging MARL methods when the action dimensionality is very high.

4.1 Pistonball

Pistonball is a PettingZoo environment [1], consisting of a ball that needs to move from right to left with the help of equidistant pistons which can go up and down. It spawns the ball at the right-most piston, and by raising and lowering the pistons, the ball moves to the other end of the map. The environment is initially multi-agent, each agent controlling a piston, but we can model it with a single one.

We can configure the environment with as many pistons as we want, increasing the difficulty as we place more pistons. Besides, one episode lasts 125 steps, terminating it abruptly if the ball does not reach the left-most position. On the other hand, if the ball gets to the end in fewer transitions, the episode also finishes. In Figure 6, we show a frame of the environment configured with 15 pistons:

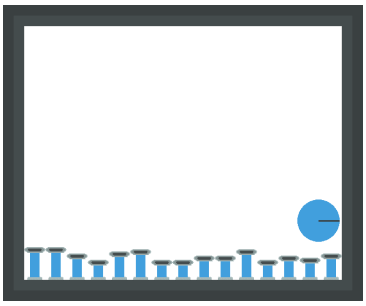


Figure 6: Pistonball initial state

The environment works with discrete and continuous actions, the action being a vector with as many positions (i.e. sub-actions) as pistons, specifying the amount we want to raise or lower each piston. In the discrete case, we consider three sub-actions for each piston: raise the piston 4 pixels, lower it 4 pixels, and don't move it. In the continuous, each sub-action takes a value in the interval $[-1, 1]$, the ends and 0 meaning the same as the discrete actions, and the values in-between raising and lowering the piston in the proportional amount.

Nevertheless, this environment has two problems we must solve. First, the observation and state space consists of images, unnecessarily complicating and slowing down training if we consider the information the environment maintains and its low complexity. As images, the state gives information about the ball position and the height of each piston. In fact, Figure 6 is the initial state of the environment.

The observation is a cropping of the state, containing the current piston and the immediate left and right neighbours (or the walls in the extremes). If the ball is not over those pistons, it does not appear. However, the cropping does not show the whole piston, so we cannot know its exact height, but we still know if we need to raise it or lower it more. In Figure 7, we show the three observation types we can have (independently of the ball):

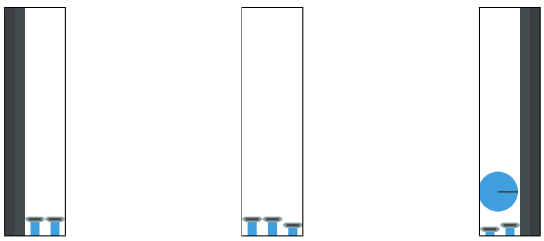


Figure 7: Pistonball observations of the leftmost, second leftmost, and rightmost pistons

[1] https://pettingzoo.farama.org/environments/butterfly/pistonball/

The environment keeps the pistons and ball xy-coordinates, and considering the downsides of training with images, we decided to redefine the observation and state space. Nonetheless, we wanted to keep the original complexity without adding extra information. Hence, the observation should only contain information about the current and neighbour pistons and the ball if it is visible. The state could contain information about all pistons and the ball.

At first, we thought about using the coordinates but found a simpler way. Moreover, using the x-coordinate would give away the piston, hindering learning a single behaviour that could scale to more pistons. In the end, we only need to know if the ball is above the current piston or its immediate neighbours, and if the piston height exceeds the ball's, we need to lower it.

Therefore, the observation is now a vector containing four boolean flags. The first three indicate if the ball is above the current, left, or right neighbour. If some neighbour is missing because there is a wall, we pass a 0. We consider the ball over a piston if any part of the ball is inside the horizontal space the piston occupies, independently of the height, similar to the original observation.

The fourth flag only applies if the ball is visible from one of those three pistons. It is true if the ball's bottom is higher than the current piston's height. We use this flag because the other three do not consider the y coordinate, and the ball could be lower than the piston, so raising it more would be counter-productive.

The state is a vector with as many positions as pistons, containing the first flag of the observation for that piston to indicate if the ball is over that piston. If the ball is over the current piston, it is always higher, so we do not need the fourth flag. And we do not need the second and third one as the state contains information for each piston, so we would end up repeating information.

The second problem is the reward function is not suitable for MARL methods factorizing the joint action-value function. It consists of two terms: a global reward given to all agents and a local one given to those helping to move the ball to the left. This local term breaks the idea of a shared reward in cooperative problems. If we already assign the contribution of each agent, there is no point in trying to decompose the joint reward using factorization methods. We could use MADDPG, which accepts different reward functions, but that is not the point of this thesis.

Thus, we removed the local term and kept the global as a shared reward function, making the task more difficult. It gives a reward equal to the percentage of distance the ball has travelled since the last step, plus a time penalty (i.e. negative term) to force moving the ball as fast as possible. Thus, the maximum episode return would be 100 without a time penalty. In our experiments, we used a time penalty of -0.25, and the maximum return we could achieve with 15 pistons was around 95∼97.

4.2 ManyAgent Swimmer

ManyAgent Swimmer is a multi-agent version of Swimmer [6] that one can configure with as many agents as desired. Nonetheless, we first need to explain how MuJoCo and the original Swimmer work and how the people of FACMAC transformed all the single-agent MuJoCo environments into multi-agent ones.

4.2.1 Swimmer

Swimmer [2] belongs to MuJoCo (Multi-Joint dynamics with Contact), a general-purpose physics engine to help research in areas such as robotics and RL. All MuJoCo environments feature a robot with some shape that needs to reach a goal. Swimmer features a worm-like robot that needs to "swim" through the floor as far as possible in the right direction. In Figure 8, we show a frame of the environment, where

[2] https://mgoulao.github.io/gym-docs/environments/mujoco/swimmer/

we can see the robot, which consists of three long segments connected by two hinge-like joints to allow rotation between them.

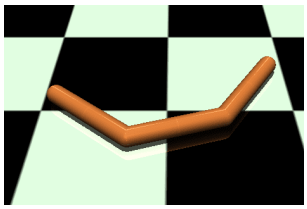


Figure 8: Swimmer frame

MuJoCo creates an XML file [3] for each environment, defining the shape of the agent and how it moves. In particular, we usually encounter the following elements:

- Multiple *body* elements defining parts of the agent's body, each with a *geom* element defining its geometry (e.g. sphere, cylinder, or capsule).
- Multiple *joint* elements defining motion degrees of freedom between the *body* containing the *joint* and the parent *body*. We can have different types, like hinges defining rotation or sliders defining translation.
- A *worldbody* element with a plane type *geom* defining the floor the agent moves in.

Hence, we usually have a nested tree structure of *body* elements, each with one or more *joint* elements defining movement between parent and child *body*. Moreover, the root is usually the *worldbody* because we can also define how the agent moves with respect to the floor using *joint* elements (e.g. the worm-like robot in Swimmer slides through the floor).

Swimmer has one *worldbody* defining the floor, three *body* defining the capsule segments, and five *joint*. Two are hinges to create rotation in the z-axis between two connected capsules, and the other three define motion between the first capsule (i.e. the front tip) and the floor. In particular, one hinge defines rotation in the z-axis, and two sliders generate the sliding movement in the x and y-axis.

Each *joint* has different information about its *body* depending on the joint type, calculated with respect to the parent *body* coordinate axes. Swimmer only has hinges and sliders. The hinges return the segment's angle and angular velocity with respect to the x-axis, and the sliders the segment's x and y position and velocity. Altogether, the state consists of ten unbounded scalars expressed in units that do not take very high values (e.g. radians).

The action space consists of the torque we can apply to the desired *joint*'s. We specify which by defining inside the *actuator* element, different *motor* elements and linking each to a particular *joint*. Swimmer only allows applying torque to two of the three hinges, the ones rotating the second and third capsule segments, but not the first one or the two sliders. Hence, the action is a 2-dimensional vector, each position taking value in the $[-1, 1]$ range.

Finally, the reward function consists of two terms, the second one subtracted from the first one. The first term gives a reward based on how much the swimmer has advanced (i.e. to the right) since the previous step, being negative if it moves the opposite way. The second term penalizes if it takes too large actions, weighted by a constant we can specify.

[3] https://mujoco.readthedocs.io/en/stable/XMLreference.html

4.2.2 Multi-Agent Swimmer

Multi-agent MuJoCo (MAMuJoCo) [4] is the multi-agent version of MuJoCo, developed by the same people of [31]. MAMuJoCo acts as a wrapper over the single-agent environments to divide the state and action space among a set of agents. Each agent controls one or multiple *joint*'s, receiving the corresponding part of the state (i.e. observation) and taking care of the corresponding sub-actions in the action vector. We can set different configurations for each MuJoCo environment, changing the number of agents and how many *joint* elements each agent controls.

Multi-agent Swimmer (not ManyAgent Swimmer) allows a single configuration of two agents, taking care of one of the two controllable hinges. The observation of each agent consists of the hinge's data (i.e. the angle and angular velocity of its segment). By default, none of them receives the information of the other hinge and the two sliders, but we can add them if needed. However, their information is still part of the fully-observable state, which remains the same. The reward function is shared among the two agents and does not change.

MAMuJoCo has a parameter to control how far an agent can observe, called *obsk*. In Multi-Agent Swimmer, if we set it to 0, each agent only observes its controllable hinge, and if we set it to 1, each would also observe the other controllable hinge. Nonetheless, they would still not see the sliders and the remaining hinge. In our experiments with all MAMuJoCo environments, we set it to 0 to simulate the situation of partial observability and no communication during execution.

4.2.3 Definition

MAMuJoCo authors also created a new multi-agent version of Swimmer, called ManyAgent Swimmer, defining a custom robot with as many segments as the user wants. Thus, we need a new XML file for each different length.

The parameter $agent\_conf$ allows us to specify how we want to configure the environment in the form of $number\_of\_agents \times hinges\_per\_agent$. Thus, if we set it to "2x2", we will create a 5-segment Swimmer with two agents controlling two hinges each. As the original version, it also defines the other three *joint*'s to create motion with respect to the floor, still not controllable. In Figure 9, we show a frame of a 10x2 Swimmer to demonstrate how we can create it as long as we want.

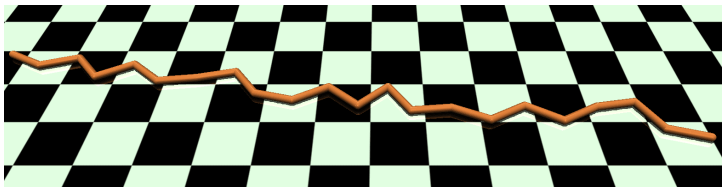


Figure 9: 10x2 Swimmer frame

The observation and action space follows the same idea as the multi-agent wrapper and divides them depending on which hinges each agent controls. Thus, in the 2x2 Swimmer, each agent applies torque to two hinges and receives those hinges' data. The fully-observable state remains the same, but now with the new hinges' information. Finally, the reward function does not change, but as the robot has more segments to move, the maximum accumulated reward one can get in an episode increases.

[4] https://robotics.farama.org/envs/MaMuJoCo/

4.3 Ant

Ant [5] is another MuJoCo environment (i.e. initially single-agent) first defined in [36]. It features a four-legged ant-like robot that has to learn to walk to the right. As another MuJoCo environment, it has an XML file defining the agent body and movement. In Figure 10, we show a frame of the environment to see the ant structure:

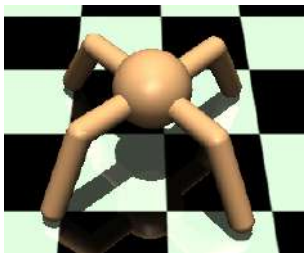


Figure 10: Ant frame

We can distinguish the torso and the four legs. The torso consists of a sphere *body* element with a *joint* of type *free*, which is a special type defining rotation and translation in the three axes, allowing it to float. Without using that particular joint type, the torso would fall. Each leg consists of mainly two *body* elements with capsule geometry and a hinge *joint* defining rotation.

In the XML file and the documentation, the leg hinges are called hip and ankle, with a sub-index indicating the leg. If we look at Figure 10, the hip is the hinge of the closest segment to the torso (the one oriented horizontally), while the ankle is the hinge of the furthest segment (the one more-or-less oriented vertically). The hip defines rotation in the z-axis, while the ankle defines it in the x and y-axis.

Nonetheless, we found it a little confusing about how the XML and the documentation name the legs if we look at the simulated environment. Each leg has an index, the same as their corresponding hip and ankle hinges. In Figure 11, we show another frame where we have recoloured the legs:

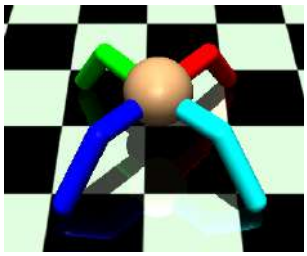


Figure 11: Recoloured Ant frame

The front left leg is red, the front right green, the back left blue, and the last cyan, the indices following the same order (i.e. *hip_1* and *ankle_1* belong to the front left leg). Thus, what is strange is that the authors changed left and right in the front and back legs, which is confusing, especially if we want to divide the environment into multiple agents.

Regarding the information from each *joint*, the hinges also return the corresponding segment's angle and angular velocity (both scalars), calculated with respect to the parent *body*. The *free joint* returns the torso's centre position, linear velocity, and angular velocity. As we have translation and rotation in all three axes, it returns them as xyz-coordinates. Moreover, it returns the torso orientation as a quaternion.

[5] https://mgoulao.github.io/gym-docs/environments/mujoco/ant/

The state is a 27-dimensional vector because it does not include the torso's x and y position by default, which we do not need. The action space is an 8-dimensional vector defining the torque we can apply to each hinge, taking values in the $[-1, 1]$ range.

Like the previous environment, one episode lasts 1000 steps. However, it can end earlier if the torso's z-position is outside a certain range. Usually, being outside that range means the ant has destabilized and is likely to fall backwards. Similar to a turtle, it cannot move when upside-down, so the episode finishes.

Finally, the reward function consists of three terms, one subtracted from the other two. Every step the ant is alive, the environment returns a reward of 1. The other two terms are the same as the previous environment, one giving a reward based on how much the ant has advanced and a penalization term for taking too large actions.

MAMuJoCo also divides this environment into multiple agents, defining different configurations with different numbers of agents, each controlling different legs. For example, we can define four agents, each controlling one leg or two agents, each controlling two [6].

Each agent is responsible for applying torque to its corresponding hinges, each receiving the information of those hinges as observation. By default, none receives the torso information, only available in the original full-observable state, which remains the same. However, we can add some information from the state if needed. The reward function does not change, but now all the agents share it.

5 Existing Methods

In this section, we will explain the MARL methods we have used in our experiments, which we already introduced in subsection 2.6, but we will complete their explanation here. All follow the CTDE framework but work with different action types and handle the joint reward differently. Moreover, in section 6, we will explain a new method we have developed based on FACMAC.

5.1 QMIX

QMIX is a value-based method working with discrete actions and decomposing the joint action-value function Q_{tot} into the individual functions $[Q_a]_{a=1}^{N}$. QMIX appears as a relaxation of VDN, another value-based method decomposing the joint function into a summation, which can be very strict and fail to solve tasks where the joint function does not accept such decomposition.

QMIX's authors saw that to extract decentralized policies consistent with the centralized training, we do not need VDN's complete factorization. We only need to ensure the global *argmax* performed on Q_{tot} yields the same result as the set of individual *argmax* performed on each individual Q_a. Hence, with a factorization that fulfils the IGM principle, each agent acts decentralized by choosing greedily with respect to its Q_a and computing the *argmax* over Q_{tot} becomes trivial.

QMIX learns a non-linear monotonic factorization using a neural network. Compared to VDN, QMIX's factorization is more relaxed and can represent any joint function that is a non-linear monotonic combination of the individual ones, including the ones VDN can represent. To ensure monotonicity, QMIX imposes a constraint on the relationship between Q_{tot} and each Q_a:

$$\frac{\partial Q_{tot}(\boldsymbol{\tau}, \mathbf{u}, s)}{\partial Q_a(\tau_a, u_a)} \geq 0 \tag{42}$$

[6] https://robotics.farama.org/envs/MaMuJoCo/ma_ant/

In Figure 12, we present QMIX's architecture. It defines one network per agent to compute its Q_a and one global mixing network to compute Q_{tot}. QMIX's authors employed a slightly different notation than the one we use, which we will follow when explaining the networks. In particular, the subscript indicates the time step instead of the agent when the two are present.

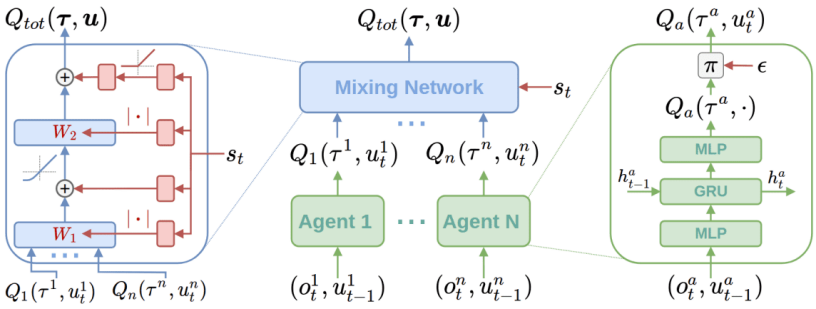


Figure 12: QMIX architecture (taken from [34])

Each agent network is a DRQN to tackle partial observability with the help of a recurrent layer. It receives as input the agent's current observation o_t^a and the previous action u_{t-1}^a, and together with the recurrent layer's previous hidden state h_{t-1}^a, the recurrent layer maintains a representation of the trajectory τ^a. The network outputs the action values for all the actions, and with ϵ-greedy, it selects the action to execute, later stored in the replay buffer.

The mixing network computes Q_{tot} from the individual $[Q_a]_{a=1}^N$ and the environment's true state s_t. It consists of a two-layer feed-forward network whose weights and biases come from four different hypernetworks, each consisting of a single linear layer receiving the state as input. The hypernetworks computing W_1 and W_2 use an absolute activation to ensure they are positive, which is necessary to fulfil Equation 42.

The true state s_t acts as input of the hypernetworks instead of the mixing network to allow Q_{tot} to depend on the state information in a non-monotonic way. We could pass it as input to the monotonic function the mixer approximates, but it would be overly constraining. By using it as input to the hypernetworks, we can condition the weights of the mixing network on s_t in an arbitrary way, integrating the state information into Q_{tot} as flexibly as possible.

We train the individual agents' networks and the mixing network to minimize the MSBE loss, computed using Q_{tot}. Hence, we use ϕ to refer to the combined parameters of all networks and the same with ϕ_{targ} and the target networks. We present the loss function below, where D is the replay buffer, and $Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}, s, \phi)$ denotes the whole computation of Q_{tot} starting from the agents' observations and previous actions, explaining why it depends on $\boldsymbol{\tau}$ and $\boldsymbol{u}$:

$$\begin{aligned} L(\theta, D) &= \mathbb{E}_{(\boldsymbol{\tau}, \boldsymbol{u}, r, \boldsymbol{\tau'}, s, s') \sim D}[(y_{tot} - Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}, s, \phi))^2] \\ y_{tot} &= r + \gamma max_{\boldsymbol{u'}} Q_{tot}(\boldsymbol{\tau'}, \boldsymbol{u'}, s', \phi_{targ}) \end{aligned} \tag{43}$$

Even though the architecture depicts a network for each agent, we can train a single one shared by all agents. In that case, we might want to add an agent identifier to differentiate the inputs of each agent such that the network can learn different behaviours if needed. If all behave identically, we do not need those identifiers.

5.2 MADDPG

MADDPG is an actor-critic MARL method that does not factorize the joint action-value function. It is the multi-agent version of DDPG, so it only works with continuous actions and learns deterministic policies. Even though we want to focus on methods factorizing the joint function, we also train our environments with MADDPG to compare and see the advantages of those methods.

Methods not factorizing the joint action-value function have disadvantages. First, it does not tackle the multi-agent credit assignment problem. LICA authors said that without an explicit mechanism like COMA, a method that optimizes the policies with a good estimation of the true joint action-value function should learn a reasonable credit assignment. However, MADDPG usually fails to learn any, resulting in worse performance than methods like COMA or those factorizing the joint action-value function.

Second, the method does not scale well because of the centralized and monolithic critics. Each receives the agents' actions and the hidden state as input. Moreover, it has to learn to generalize through all the agents' actions. Hence, MADDPG usually fails to scale to environments with lots of agents or those where it has to explore a lot of the continuous interval each sub-action consists of.

Besides, with those monolithic critics, we cannot learn a task, increase the number of agents, and reuse the learned weights in the new task. Each receives the actions of each agent. Hence, if we increase the number of agents, the input size changes, and we have to retrain the centralized component again and maybe the decentralized actors.

We present MADDPG's architecture in Figure 13. For each agent, MADDPG learns a decentralized policy and a centralized critic. It trains multiple critics because MADDPG works with competitive tasks where each agent has a reward function. However, in cooperative problems, all the critics will learn to approximate the same joint action-value function. Hence, we could train a single critic, still receiving all the agents' actions.

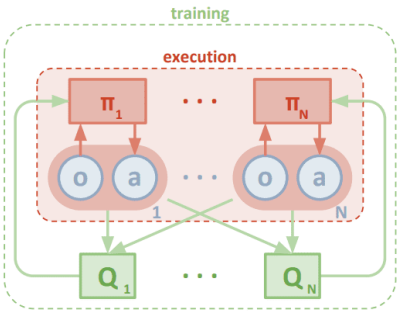


Figure 13: MADDPG architecture (taken from [24])

The actor networks consist of MLPs instead of using some recurrent layer to handle partial observability. Thus, each policy $\pi_a(o_a^t, \theta_a)$ is parameterized by the agent's observation o_a^t instead of the trajectory τ_a. Nonetheless, some implementations like the one in [30] consider using a DRQN. As with QMIX, we can train a shared policy if all behave identically or try to add agent identifiers if not.

Likewise, each centralized critic consists of an MLP, learning to approximate the same joint action-value function using all agents' actions and the hidden state. The state grants complete observability, so we do not need any recurrent layer. If the state were unavailable, we could use the observations and still forget about a recurrent layer as long as the observations do not miss vital information from the state.

We train each centralized critic to minimize the MSBE, sampling transitions from the replay buffer. Like DDPG, we do not need to compute an exhaustive maximum, as the joint policy approximates the action with the maximum action value. Moreover, like DDPG, it samples the target actions from the current target policies. In particular, the critic loss is as follows:

$$\begin{aligned} L(\phi_a, D) &= \mathbb{E}_{(s,\boldsymbol{u},r_a,s')\sim D}[(Q_a(s, u_1, \dots, u_N, \phi_a) - y)^2] \\ y &= r_a + \gamma Q_a(s, u'_1, \dots, u'_N, \phi_{targ_a})|_{u'_j=\pi_j(o_j,\theta_j)} \end{aligned} \tag{44}$$

To train each policy, it directly maximizes the action-value function of its critic, like DDPG. However, it samples the actions of the other agents from the replay buffer, which can cause problems because the other agents' policies may have drastically changed since those actions were stored. In particular, the actor objective function is as follows:

$$J(\theta_a, D) = \mathbb{E}_{(s,\boldsymbol{u}_{-a})\sim D}[Q_a(s, u_1, \dots, u_N, \phi_{targ_a})|_{u_a=\pi_a(o_a,\theta_a)}] \tag{45}$$

Even though MADDPG should only work for continuous actions, the authors used it with discrete ones thanks to the Gumbel-Softmax trick [18, 44], which does the same as the reparametrization trick in SAC but with a categorical distribution instead of a Gaussian. Now, the actor generates logits instead of actions (i.e. we do not apply any activation function like *tanh*), and we sample actions from it. Nonetheless, it updates the networks with the same loss functions.

5.3 FACMAC

FACMAC is an actor-critic method working with environments with continuous actions, factorizing Q_{tot} using a mixing network like QMIX. However, thanks to being an actor-critic method, it can also learn non-monotonic factorizations, which QMIX could not. We can see FACMAC as mixing MADDPG and QMIX to solve tasks we cannot solve with monolithic or monotonically factored critics. In Figure 14, we present FACMAC's architecture:

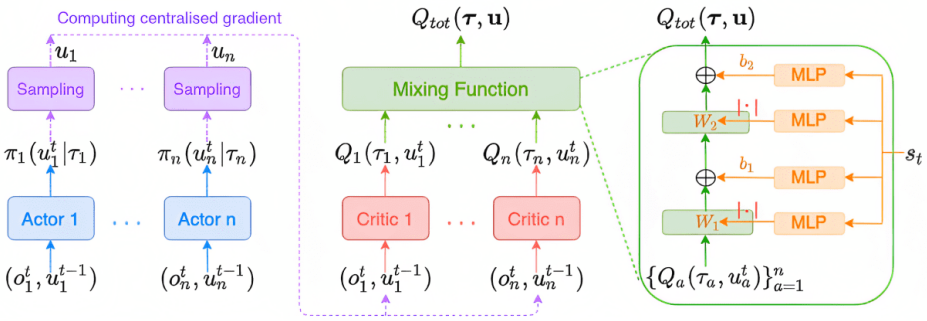


Figure 14: FACMAC architecture (taken from [31])

Like MADDPG, we have an actor for each agent to approximate a deterministic policy. Figure 14 depicts the policy network as a DRQN because it receives the current observation o_a^t and the previous action u_a^{t-1} to maintain a representation of the current trajectory τ_a with the recurrent layer hidden state, explaining why the policy depends on τ_a at the output.

Nonetheless, the authors also experimented with some environments training an MLP that only receives the observation as input. Besides, like the previous methods, we can share weights and train a single network, adding agent identifiers to learn different behaviours.

Unlike MADDPG, the individual critics do not compute Q_{tot} but Q_a like QMIX, each receiving the agent's observation and previous action. For the same reasons as with the actor, the critic architecture depicted in Figure 14 seems to be a DRQN. However, the authors also experimented with environments where an MLP, only receiving the observation as input, was a better option. Besides, we can train a single shared critic instead of one per agent, adding agent identifiers to learn different action-value functions.

The mixing network has the same architecture as QMIX. However, thanks to using an actor-critic architecture and not constraining the critic design, FACMAC can learn any factorization without imposing constraints, including non-monotonic ones. Figure 14 depicts a monotonic factorization because it uses absolute activations to generate W_1 and W_2. However, the authors also experimented with some environments where removing those activations and learning a non-monotonic decomposition achieved better results.

FACMAC updates the critics and the mixer using the same loss as QMIX (i.e. the MSBE computed using Q_{tot}). Moreover, it maintains a target network for each critic and the mixing network to avoid the moving target problem. We present the loss function below, where ϕ represents the parameters of the critics and the mixer. $Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}, s, \phi)$ denotes the whole computation of Q_{tot} starting from the agents' observations and previous actions, explaining why it depends on $\boldsymbol{\tau}$ and $\boldsymbol{u}$:

$$\begin{gathered} L(\phi, D) = \mathbb{E}_{(\boldsymbol{\tau}, \boldsymbol{u}, r, \boldsymbol{\tau'}, s, s') \sim D}[(y_{tot} - Q_{tot}(\boldsymbol{\tau}, \boldsymbol{u}, s, \phi))^2] \\ y_{tot} = r + \gamma Q_{tot}(\boldsymbol{\tau'}, \pi(\boldsymbol{\tau'}, \theta_{targ}), s', \phi_{targ}) \end{gathered} \tag{46}$$

FACMAC trains the policy to maximize Q_{tot}. However, it modifies MADDPG's loss to solve the problems leading to converge to sub-optimal policies. It trains the policies together instead of computing a different policy gradient for each agent, sampling all the actions from their respective policies instead of some from the replay buffer. Besides, training them together as a single joint policy leads to learning more coordinated behaviour. In particular, the actor objective function is as follows, where θ represents the parameters of all actors:

$$J(\theta, D) = \mathbb{E}_{(\boldsymbol{\tau}, s) \sim D}[Q_{tot}(\boldsymbol{\tau}, u_1, \ldots, u_N, s, \phi)|_{u_j = \pi_j(\tau_j, \theta_j)}] \tag{47}$$

5.4 TransfQMIX

TransfQMIX is a value-based method working with discrete actions and factorizing Q_{tot}. It improves QMIX by replacing the agent and mixer networks with transformers. Previous methods mainly focus on how to factorize the joint action-value function without worrying about the agent and mixing networks architecture and how to build the observations and the environment's true state.

The authors observed that we could use the features describing a coordination problem as vertex features of a latent graph. Thus, TransfQMIX uses transformers to leverage this latent structure and learn better coordination policies. The transformer agents perform graph reasoning over the observable entities' state, and the transformer mixer learns a monotonic function from a larger graph that includes the agents' internal and external states.

Previous methods build their observations and state by simply chaining features for the different entities we encounter in the environment. For example, in the pistonball environment we have defined, the entities are the pistons (we do not count the ball), and we obtain four flags from each. The problem with chaining features is that we process the same feature type with different weights because they belong to different entities.

For this reason, TransfQMIX generalizes a set of features to describe the state of the entities observed by the agent or from a global point of view. Thus, we need to keep a set of features available for all the

observed entities that will represent the vertices of the latent coordination graph. In particular, each agent observes at each time step k entities, and a set of z features defines each entity. Each feature may take different values for each agent because of the partial observability.

Defining entity i observed by agent a at time step t as $ent^a_{i,t} = [f_1, \ldots, f_z]^a_{i,t}$, TransfQMIX replaces the usual observation vector with the observation matrix with dimension $k \times z$ that includes all the k entities agent a observes at time t:

$$O^a_t = \begin{bmatrix} ent_1 \\ \vdots \\ ent_k \end{bmatrix}^a_t = \begin{bmatrix} f_{1,1} & \ldots & f_{1,z} \\ \vdots & \ddots & \vdots \\ f_{k,1} & \ldots & f_{k,z} \end{bmatrix}^a_t \tag{48}$$

This matrix structure allows processing the same feature type using the same weights of an embedding matrix Emb^O with shape $z \times h$, where h is the embedding dimension. The resulting matrix $E^a_t = O^a_t Emb^O$ consists of the vertex embeddings $[e_1, \ldots e_k]^{aT}_t$ later processed by the transformer. The good point is that the embedding matrix is independent of k, making the approach more scalable and transferable in the number of entities than using the observation vectors that would require $k \times z \times h$ parameters to encode the features.

One problem with the embedding matrix is that we lose positional information on the different features because we process each type with the same weights. A traditional encoder could learn that features at a specific location are relevant to some entity and need special treatment. Moreover, encoding one-hot features (e.g. the agent identifier) is not direct with the observation matrix. Consequently, the authors created two additional binary features to compensate for those drawbacks. The first feature is *IS_SELF*, which informs if the entity is the agent to which the observation matrix belongs:

$$f^a_{i,IS\_SELF} = \begin{cases} 1, & if\ i = a \\ 0, & otherwise \end{cases} \tag{49}$$

It will be 1 for $ent^a_{a,t}$ and 0 for the others. We can see this feature as the re-adaptation of the agent identifier. The second feature is *IS_AGENT* and tells if the entity is a cooperative agent or not, allowing the encoder to treat the features of the agents differently. Assuming A is the set of agents:

$$f^a_{i,IS\_AGENT} = \begin{cases} 1, & if\ i \in A \\ 0, & otherwise \end{cases} \tag{50}$$

TransfQMIX applies the same formulation with the global state, assuming the same set of z features for each of the k entities, even though we could use a different one. For example, adding IS_SELF does not make sense. Hence, the method defines the state matrix S_t with dimensions $k \times z$, defining the vertex features of all entities from a global point of view (i.e. its true value):

$$S_t = \begin{bmatrix} ent_1 \\ \vdots \\ ent_k \end{bmatrix}^a_t = \begin{bmatrix} f_{1,1} & \ldots & f_{1,z} \\ \vdots & \ddots & \vdots \\ f_{k,1} & \ldots & f_{k,z} \end{bmatrix}^a_t \tag{51}$$

Likewise, we can process the state matrix with an embedding matrix Emb^S, obtaining the embedding vertices later processed by the transformer $E_t = [e_1, \ldots, e_k]^T_t = S_t Emb^S$. In Figure 15, we present TransfQMIX's architecture:

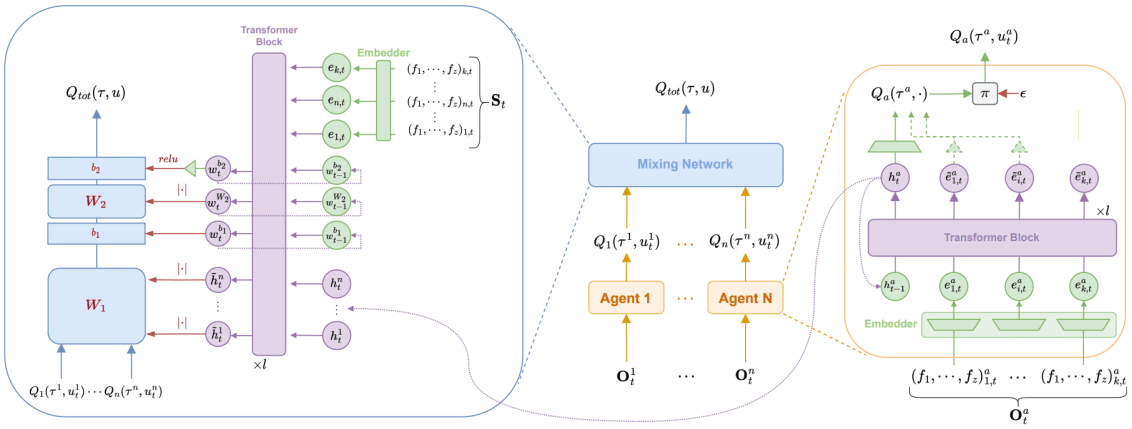


Figure 15: TransfQMIX architecture (taken from [14])

The transformer agent takes as input the matrix $X_t^a = [h_{t-1}^a, e_{1,t}^a, \ldots, e_{k,t}^a]^T$ that represents the initial coordination graph. It consists of the embedded vertices E_t^a plus a recurrent hidden vector h_{t-1}^a with size h initialized with 0s. The transformer output $\hat{X}_t^a$ is a refined graph in which the vertices are altered based on the attention given to the others. Thus, multi-head self-attention learns the edges of the coordination graph.

The hidden state acts similarly to the [CLS] token in BERT, and the transformed vector $\hat{h}_{t-1}^a$ encodes the general coordination reasoning of agent a. Thus, the agent samples the action from this hidden state using a feed-forward layer. $h_t^a = \hat{h}_{t-1}^a$ is passed to the next step, allowing the agent to update its coordination reasoning recurrently.

TransfQMIX also uses an MLP to compute Q_{tot} from $[Q_a]_{a=1}^N$. However, it does not generate the weights and biases from separate hypernetworks but from a transformer. The transformer mixer receives as input the coordination graph $X_t = [h_t^1, \ldots, h_t^N, w_{t-1}^{b_1}, w_{t-1}^{W_2}, w_{t-1}^{b_2}, e_{1,t}, \ldots, e_{k,t}]^T$. $h_t^1, \ldots, h_t^N$ are the hidden states of the agents, $w_{t-1}^{b_1}, w_{t-1}^{W_2}, w_{t-1}^{b_2}$ are three recurrent vectors initialized with 0s to learn the MLP weights, and $e_{1,t}, \ldots, e_{k,t}$ are the state embedded vertices.

This larger coordination graph enters the transformer blocks, and again, the transformer creates a new graph $\hat{X}_t$ by refining the vertices based on the attention put on the others. $\hat{h}_t^1, \ldots, \hat{h}_t^N$ are the agents' coordination reasoning enhanced with global information they had no access to (i.e. the hidden state of the other agents and the environment's true state). The mixer uses these refined vectors to build W_1 by applying linear layers with an absolute activation.

Considering how each agent transformer generates the corresponding Q_a from h_t^a, and then the mixing network multiplies it by W_1, generated from the refined $\hat{h}_t^a$, the mixer's goal is combining and refining the independent agents' reasoning so that each one represents the team coordination to learn Q_{tot}.

The authors generate W_2, b_1 and b_2 by applying linear layers to the recurrent vectors. To adhere to QMIX's monotonicity constraint, the linear layers generating the weight matrices use absolute activations. The authors use recurrent vectors to incorporate temporal dependence on the centralized training as they argue that Q_{tot} heavily depends on prior states, and we should incorporate this reliance explicitly in the mixer network.

Besides, using recurrent vectors makes the mixer network (the MLP, not the transformer) inde-

pendent of the number of entities. If Q_A represents the vector containing the individual action-value functions, we can compute Q_{tot} as follows. In the equation, we show the dimension of each vector and matrix, and we can see how none of them depends on k.

$$Q_{tot} = (Q_A^{(1\times N)} W_1^{(N\times h)} + b_1^{(1\times h)}) W_2^{(h\times 1)} + b_2^{(1\times 1)} \tag{52}$$

Together with the embedding matrices, which are also independent of k, TransfQMix is transferable because the networks' weights constitute an attention mechanism independent of the number of vertices. Hence, we can apply the same parameters to evaluate and train the same environment with more or fewer agents. Conversely, we must re-train a traditional model every time we introduce a new entity because the inputs to the network change size, having to adjust the networks' weights.

6 JAD3

We developed *Joint Action-Value Decomposition with TD3* (JAD3) using FACMAC as our basis. We discovered FACMAC when searching for some method factorizing the joint action-value function that could work with continuous actions. We found others that supposedly could tackle those actions, but none demonstrated it with results. FACMAC presented results, but we could not reproduce them exactly with their code, even obtaining opposite results in some cases.

Besides, we did not understand why factorizing the joint value function if they still use it to update the actor. They lose the advantage of learning individual value functions representing each agent's contribution to the joint reward, allowing them to update each policy accordingly. In case all agents behave and contribute identically, it might not make a difference, but we believe it will in case they do not.

Using the joint value function does not allow to know which agent executed which action, which can cause wrong policy updates for some agents that could give too much importance to the wrong action. Maybe we are getting a high reward thanks to an agent's action, but another agent executed the wrong one. In that case, we would still give more probability to that wrong action. Moreover, as FACMAC learns a deterministic policy, it will always execute that wrong action.

We believe the authors started from MADDPG and tried to improve the flaws in the actor loss function we have explained before, resulting in the equation FACMAC uses. The article does not mention the possibility of using the individual functions to update the actor, so we believe they did not explore it. Still, they affirm that learning individual critics allows for a more flexible factorization as the critic design is not constrained compared to value-based methods like QMIX.

Nonetheless, if we remember what we explained when explaining LICA, its authors affirmed that we could still achieve optimal performance without any credit-assignment mechanism (explicit or implicit) but by using the joint function to update the actor. LICA obtained better results in some discrete tasks than methods like COMA or QTRAN. Hence, even though we still thought it should work better using the individual functions to update the actor(s), we allowed this new method to use both options and conclude which works better in our experimentation.

Another change to FACMAC is adding the modifications TD3 added to DDPG to reduce the overestimation problems. Thus, we maintain two approximations of the action-value function, computing the minimum as the target to update them, adding Gaussian clipped noise to the actions coming from the target policy, and updating the policy and target networks at a lower frequency than the value functions.

The problem with two action-value function approximations is that we now distinguish between individual and joint value functions. We asked ourselves if we should duplicate the critic, the mixer, or both. However, considering how the mixer uses the critic's output as input and we update both with

the same loss function, the only option was duplicating both.

If we back-propagate the gradient and have a single mixer, that mixer might overestimate the true joint value function. In that case, those overestimated values will back-propagate through the two critics, overestimating them. On the other hand, if we keep one critic but two mixers, one mixer might overestimate some action values that will back-propagate through the critic. Even though the other mixer does not overestimate them, it will use that critic to generate the joint values, overestimating them anyway.

Another modification to FACMAC and TD3 is how we generate the noise we add to the actions, both from the exploratory and target policy. We decrease the Gaussian's σ as training proceeds, allowing for a bigger noise at the beginning and decreasing it until a minimum value. With that, we visit a wider spectrum of the action space at the first stages of training and execute better actions as the policy improves and converges.

Finally, we also changed, compared to TD3, how we update the actor. TD3 maximizes the action values from the first approximation instead of using the minimum. The authors do not say anything about it, but we did not find any demerit to use it like SAC does, which we found essential to obtain better results than FACMAC. Thus, we use the minimum of both approximations, but we can still compute it using the individual or joint value functions.

We present the new method's architecture in Figure 16. We distinguish between training and execution, even though one includes the other. The training section corresponds to the centralized process where we update the networks using all the agents and state information, and the execution section is the decentralized process where each agent acts using their observations and policy:

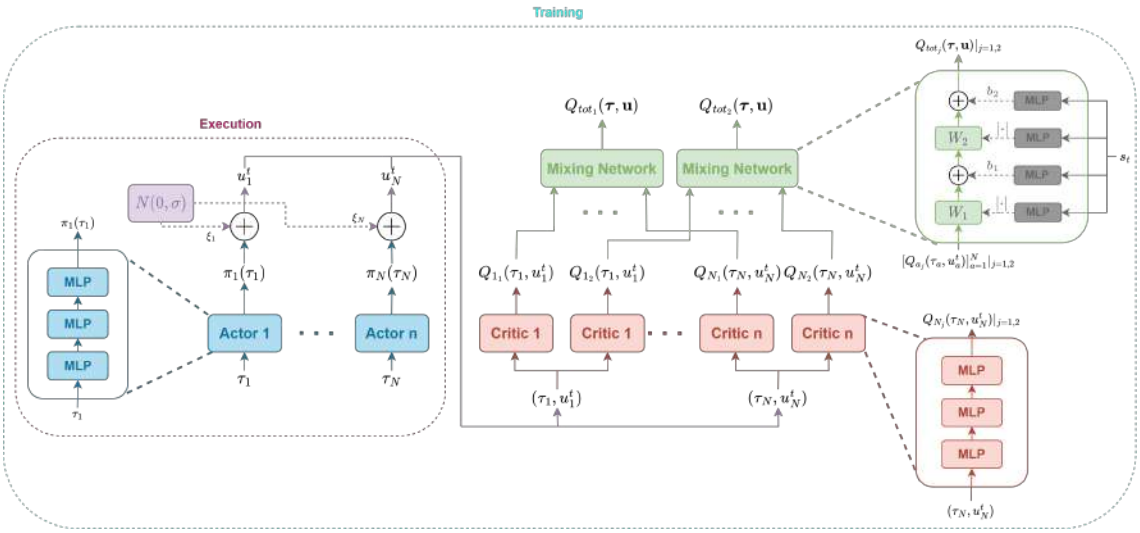


Figure 16: JAD3 architecture

In the decentralized section, we have multiple actors depending on how many different networks we want to train (i.e. if all agents behave identically or not). Each actor receives a trajectory as input and generates one action (i.e. deterministic policy). During training, we add Gaussian noise to that action to allow for exploration, but we do not add it when evaluating the learned policy (i.e. execution).

The actors' architecture is a three-layer MLP. We do not use recurrent layers because, even though they should help with partial observability, we have not observed a significant performance difference

compared to the increased execution time. Nonetheless, to leverage previous information, we allow to pass the current observation and as many previous observation-action pairs as desired. With that, the input can be an observation or a full trajectory.

The training section contains two duplicates of each critic and the mixer, even though the architecture in both duplicates is the same. We can have as many critics as agents, depending on if we need different individual functions because each agent contributes differently to the reward. Both duplicated critics receive the same input, but each mixer receives a different set of individual value functions from the different critics' duplicates.

The critics' architecture is also a three-layer MLP, applying the same trick to leverage previous information but also receiving the current action from the corresponding actor. The mixers' are the same as QMIX, learning a monotonic factorization. Even though we could learn other factorizations thanks to learning the critics, even non-monotonic, we did not need them in our environments.

Let's explain how we update the networks, starting with the critics and mixers. As in TD3, we update each approximation independently, using the minimum of both as targets. We train the mixer and critics together, and as in previous MARL methods, we use the joint action-value function to compute the MSBE loss, which we want to minimize. We present it below:

$$\begin{gathered} L(\phi_j, D) = \mathbb{E}_{(\boldsymbol{\tau}, \boldsymbol{u}, r, \boldsymbol{\tau'}, s, s') \sim D}[(y_{tot} - Q_{tot_j}(\boldsymbol{\tau}, \boldsymbol{u}, s, \phi_j))^2] \\ y_{tot} = r + \gamma min_{k=1,2} Q_{tot_k}(\boldsymbol{\tau'}, \pi(\boldsymbol{\tau'}, \theta_{targ}) + \xi, s', \phi_{targ_k}) \\ \xi \sim clip(N(0, \sigma), -c, c) \end{gathered} \tag{53}$$

We distinguish the two approximations as $j = 1$ and $j = 2$. As we train the critics and mixer together, the set of weights ϕ_j represents the weights of the critics and mixer of each approximation j (compared to ϕ_{a_j}, the weights of critic a, approximation j). Therefore, the function $Q_{tot_j}(\boldsymbol{\tau}, \boldsymbol{u}, s, \phi_j)$ includes the computation of the individual and joint function (i.e. both critics and mixer), not only the joint one. Moreover, we clip the target actions into the $[-c, c]$ range.

For each actor a, we maximize the minimum of the two action-value function approximations. However, we can use the individual functions or the joint ones. Assuming we update each actor with is corresponding Q_a, the actor objective function is as follows:

$$J(\theta_a, D) = \mathbb{E}_{\tau_a \sim D}[min_{j=1,2} Q_{a_j}(\tau_a, \pi(\tau_a, \theta_a), \phi_{a_j})] \tag{54}$$

7 Experiments & Results

In this section, we will explain our experiments and their results. We want to demonstrate the advantages of MARL methods to solve an environment with large action dimensionality, especially those with continuous actions. Second, we want to demonstrate how MARL methods factorizing the joint action-value function work better than those that do not and those training decentralized. Third, we want to prove how our new method improves FACMAC. Finally, we want to show how MARL methods can learn more general policies to solve similar tasks.

We will divide this section into the different environments we have executed, even though we might want to demonstrate the same in more than one. Within each environment, we will explain the best results of each method and then compare their results. As one of the advantages of MARL methods is the policy reusability, we will also give a section to demonstrate it in those environments we can.

Out of the four environments we experimented with, we used the first three to demonstrate the advantages of MARL methods over single-agent ones. We also used them to demonstrate how MARL

methods factorizing the joint action-value function work better than the other types. Finally, in those environments where the actions are continuous, we studied which option to update our new method is better and demonstrated how it works better than FACMAC thanks to the TD3 improvements.

We used the last environment to demonstrate how we can learn more general policies with MARL methods compared to single-agent ones. We did not experiment with all methods, only our new one and TD3. The idea was to learn a policy that could solve at the same time two similar tasks leveraging the symmetry of dividing into multiple agents, compared to TD3, which does not have this symmetry and should have a hard time learning the two tasks altogether. Nonetheless, we could not reach conclusive results.

We have executed the same method five times with each environment, setting different seeds to the random number generator (RNG) of the Python libraries we can (e.g. PyTorch and NumPy) to generate different initial training conditions to make sure the method could learn and was not by pure luck of a particular seed. Besides, setting the seeds ourselves allows one to reproduce our results.

The environments also have an RNG to generate different initial conditions for each episode. When training starts, we set an initial seed to ensure reproducibility, but each episode starts differently. With that, we ensure the method does not overfit to a particular episode. Nonetheless, those episodes, which we call training episodes, are episodes we run to gather data to update the networks, using some exploration mechanism like ϵ-greedy. Hence, we do not execute the real action the policy would output but a different one.

For this reason, to assess the real performance of the method, after a predetermined number of training steps (i.e. interactions with the environment), we perform a different set of episodes, which we call testing episodes, where the policy outputs its true action. We always execute the same testing episodes by setting the same seeds to the environment's RNG each time an episode starts. Nonetheless, we use different copies of the environment to avoid messing with the RNG of the training episodes.

We have generated plots for the training and testing episodes, even though we mostly show the testing plot. Each will show the results across the five times we train a method, which we will call runs, also presenting an average over those five runs, which we refer to as mean.

In the training plot, each point in each run represents the average return of all the training episodes completed in a predefined amount of training steps (e.g. 5,000). On the other hand, each point in the testing plot represents an average of the fixed number of test episodes (e.g. 25) we perform after a predetermined number of steps (e.g. 20,000). We compute the mean results by averaging the same points of the five runs.

Usually, both plots are different, especially at the beginning of training, because of the exploration mechanism we use in the training episodes. Nonetheless, as training proceeds and the exploration decreases, the plots resemble more as the policy outputs the true action more often. Nonetheless, it depends on the type of action. With discrete ones, there is more difference as with ϵ-greedy, we select different actions, but with continuous ones, as we slightly modify them with Gaussian noise, the plots are more similar.

We want to mention that we will not present the extensive hyper-parameter study we have performed with each method and environment, and we will directly show the best configuration we have obtained. Nonetheless, we will explain those parameters that had some relevant performance effect. Those we do not mention mean we used their default value, which we explain in Appendix A.

Moreover, to compare how the methods solve the tasks, we will render the same testing episode

with the best policy learned by each method and try to show the episode frames in Appendix B to give a view of the behaviours the different methods have learned. Nonetheless, as there are environments where the frames do not allow us to see the behaviour clearly, we will also upload all those renderings as GIFs in our GitHub repository.

Finally, we want to explain how our training loop works, as depending on the network architecture, we have trained the methods differently. We use one or more parallel environments to run training episodes to gather data to update the networks. Then, after completing one episode with each parallel environment, we store the data in the replay buffer and sample some mini-batches to update the network.

Methods relying on a recurrent layer must process the episode transitions sequentially to update the memory cell of the recurrent layer, both when exploring the environment and updating the networks. Thus, we must store the complete episode sequentially in the buffer and sample mini-batches of those full episodes to update the networks.

On the other hand, methods not using recurrent layers store individual transitions without caring about the order and do not need to complete an episode before updating the networks. They sample mini-batches of random transitions to update the networks. Nonetheless, we decided to update the networks after completing the episodes to simplify the code, as the methods can still learn optimally.

Another difference of each method type is that methods sampling random transitions sample as many mini-batches as transitions ran in the environment, compared to those sampling full episodes, which sample a mini-batch per completed episode.

In most cases, we have only used one environment to gather data. We have leveraged the BSC POWER9 cluster to perform our experiments, running jobs that can last at most 48 hours. For this reason, we have only used more than one environment in cases where we had to execute too many training steps that would exceed the job's time limit.

The problem with running parallel environments is that we update the networks after running the episodes. Hence, if we use eight parallel environments, we run eight episodes with the same policy, filling the replay buffer with similar data. In that case, we might need to increase considerably the buffer capacity to learn the optimal policy.

7.1 Pistonball Discrete

We can configure this environment with as many pistons as desired, but as no other paper used it to demonstrate their results, we used it as default with 15 pistons. In the discrete version, each sub-action can take three values. However, we removed the action of not moving the piston to facilitate learning with DQN and trained all methods with two sub-actions per piston. Even though that action should be useful, it is better to compute the maximum among 2^{15} values than 3^{15}.

One episode lasts 125 steps, ending earlier if the ball reaches the end. All MARL methods working with discrete actions rely on a recurrent layer to implement their networks, so we update those networks by sampling full episodes. DQN, on the other hand, leverages an MLP because it assumes complete observability, sampling random transitions. Moreover, as we needed a few steps to train the methods, we only used one parallel environment to gather data.

We believed all the pistons should behave identically, even those at the extreme that only see a neighbour. Each piston should lift when the ball is over them (or at its left corner) to create momentum for the ball to move to the left. With that in mind, we trained all MARL methods with a single network to approximate the action-value function, shared among all agents to select their actions (or a single

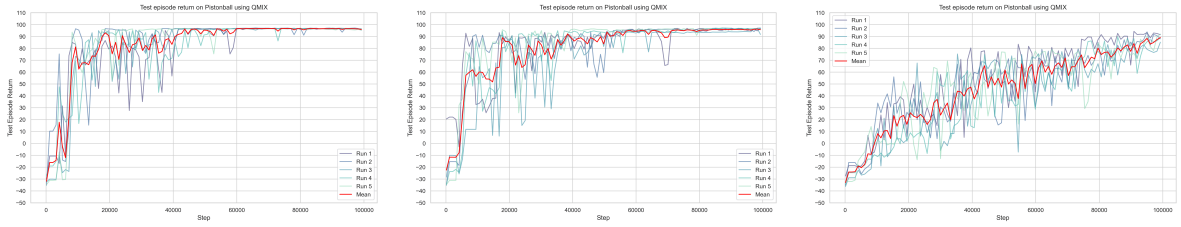
actor in MADDPG).

To further demonstrate the pistons' identical behaviour, we also trained QMIX and VDN with multiple agent networks, expecting similar or worse results than when training a single one. In particular, we trained one network per agent and separate ones for the extreme pistons. We selected QMIX and VDN because they are simple value-based methods that should take less time to train than others like MADDPG or TransfQMIX.

One could ask why we are not using more reliable methods like QTRAN or QPLEX. However, and we will later see, this environment does not have a complex factorization of the joint action-value function, and even a simpler method like VDN can solve it perfectly. Additionally, this environment has discrete actions, which is not our main focus, but we still use it to explore the advantages of MARL methods.

7.1.1 QMIX

We used EPyMARL [7] implementation to run this method. In Figure 17, we show the results obtained after optimizing the hyperparameters with the three architectures we explained:



(a) Shared networks

(b) Three networks (extremes + middle pistons)

(c) Independent networks

Figure 17: Pistonball discrete - QMIX

Among the hyperparameters we tried, only three are worth mentioning. First, we do not need agent identifiers when sharing a network, confirming the pistons act identically. Second, we obtain better results if we feed the previous action with the current observation to the action-value network. The recurrent layer stores information on previous observations and actions, such that we generate the next action using trajectories τ_a, as we can see in Figure 12.

Third, we obtain better results if we decrease the ϵ in the ϵ-greedy exploration to the minimum value of 0.05 in a few steps. We obtained all three results by decreasing it in 30,000 steps, executing the greedy action from that point in most cases. We only have two actions, so we needed less exploration.

We reached the same conclusions in all methods, especially with the identifiers and feeding the previous action. With the steps to decrease ϵ, we found, in all cases, that it was better to decrease it in fewer steps than the total training steps, but some methods needed more exploration, and we had to decrease it in more steps.

Other parameters like the learning rate, the buffer size, and the gradient clipping value had less effect. We obtained these results using the default Adam learning rate (i.e. 0.001), a buffer size of 1,000 episodes (out of the around 2,500 we execute in 100,000 steps), and clipping the gradient norm when it was over 50. We found these parameters worked with most methods, but we will mention explicitly if another value was better.

[7] https://github.com/uoe-agents/epymarl

Regarding the results, we can see how training a single network to control all the agents works better than the other two. From step 60,000 onwards, we are reaching the maximum episodic return in all five runs, even though we have slight decays, later recovered. Moreover, in all runs, the performance suddenly grows when the network grasps how to move the pistons when the ball approaches.

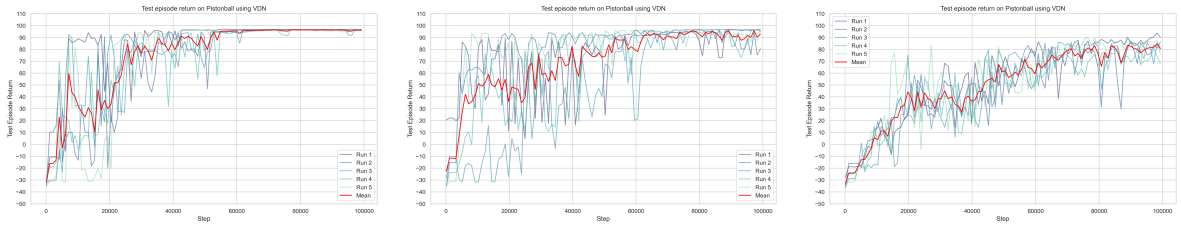
When training three networks, two controlling the extreme pistons, the results are similar but still worse, as the networks controlling the extreme pistons have seen less, and less variated, data than those controlling the medium pistons, having a harder time learning to control those pistons. Nonetheless, we still achieve a very high return in all runs, in some cases even optimal, and the results are quite stable between the runs.

Finally, training independent networks gets the worst results, even though they still learn to move the pistons decently. We have the same problem as the previous case, but worse, as each network only sees data from the piston it controls, resulting in slower but continued learning. It does not achieve optimal results, but we believe it would if we train for more steps. Regarding stability, we can see how the different runs oscillate a lot, even though those oscillations reduce as learning progresses.

We show the simulation of the first testing episode in Figure 53 using the best policy when training a single shared network. We can see how it completes it in 13 steps (frame 0 is the initial episode state) thanks to the four right-most pistons, which coordinate their movement for a few steps, generating enough height difference to create momentum for the ball to move fast and reach far without the other pistons' help. Nonetheless, some in the middle and left section must raise a little when the ball passes to keep it moving until the end.

7.1.2 VDN

We did not explain VDN in section 5 because it follows the same ideas and implementation as QMIX, only changing how we decompose the joint action-value function. We do not need any mixer network, as we decompose it as a summation of the individual action values. Regarding the hyperparameters, we needed more exploration when training more than one network, reducing the ϵ in 50,000 steps instead of 30,000. We also used a lower gradient clipping value of 0.5. In Figure 18, we show the best results we obtained:



(a) Shared networks

(b) Three networks (extremes + middle pistons)

(c) Independent networks

Figure 18: Pistonball discrete - VDN

Again, training a single shared network works better than the other two, reaching optimal performance in all runs in less than 60,000 steps. If we compare it with the same results of QMIX, VDN needs more time to learn, as the mean line does not have that sudden growth and needs more time to surpass the 80-return mark. Besides, we can see bigger oscillations in some runs until they learn. Nonetheless, we still achieve optimal performance with a simpler factorization.

If we look at the results of training three networks, they are worse than the same in QMIX, especially at the end, as some runs still fluctuate without achieving optimal performance. Moreover, the mean

line reaches a return of 90 in more than 60,000 steps compared to QMIX, which needs less than 40,000.

Finally, the results of training independent networks seem slightly better until 80,000 steps, as the mean line is higher and reaches a return of 80 faster. From that point, the line plateaus at 80 episodic return while QMIX continues to improve. However, we believe VDN can continue improving if we let it for more steps, as some runs seem to keep improving.

The worse performance, or at least a slower learning pace, is due to VDN's strict factorization, which computes Q_{tot} by adding the individual Q_a. As it calculates the MSBE loss using Q_{tot}, it will update the agent networks to output higher Q_{tot} for the joint actions that achieved a higher accumulated shared reward. However, the gradient each network will receive will be similar because of the summation, and lower, especially if the number of agents is high.

The factorization cannot shift to focus on a particular agent that might be contributing more at the beginning and learning its behaviour faster, as QMIX does. For example, the rightmost pistons will be visited more at the beginning when they must learn to start moving the ball. However, with that strict factorization, we will update all the networks at a similar pace because the gradients will be similar.

When some pistons share a network, the network starts learning slower than QMIX because not all the actions are good. As it will receive similar gradients from each agent, the worst actions will probably mess up the parameters and increase the action values for bad actions. However, as training proceeds and the network starts to approximate the action values better, even though it receives similar gradients from all agents, as they will act more optimally, it will suffer less from such strict factorization.

When we train independent networks, we suffer less from that factorization at the first stages of training, as each network focuses on learning the behaviour of a piston. On the other hand, QMIX suffers more at the beginning because it shifts its decomposition on some pistons first. Hence, the networks controlling the other agents will probably learn slower, acting worse, and the episodes end with a lower return.

We simulated the testing episodes to see if VDN learned the same coordinated strategies as QMIX. Overall, we saw VDN solving them almost identically as QMIX, even though, in some cases, one method or the other spent one or two extra steps because of a piston moving slightly differently.

7.1.3 IQL

Independent Q-learning (IQL) is a value-based method following the naive approach of decentralized training and execution, ignoring the existence of the other agents. Thus, we cannot learn any decomposition of the joint action-value function, and each agent's network learns to approximate it using that agent's information (i.e. without sharing them with the others).

However, even if we ignore the other agents, the shared reward will change from all the agents' actions, suffering from non-stationarity when trying to approximate the joint function. As it decides the action to execute based on Q_{tot}, it might wrongly update the network because it might think its action produced a high return when it was due to another agent's action.

Each agent updates its network independently, computing the MSBE loss like in DQN but using Q_{tot} instead. However, we can train a shared network if all agents behave identically, updating it by computing the MSBE of all agents and averaging the results. We have not explained this method either in section 5 because we can implement it by using QMIX as the basis and removing the mixer network, which we have done.

We have demonstrated in QMIX and VDN that we obtain better results when training a shared network without identifiers, meaning the agents behave identically. Hence, in all the remaining methods, we only trained a shared network. However, we still did a hyperparameter study. In Figure 19, we show the best results:

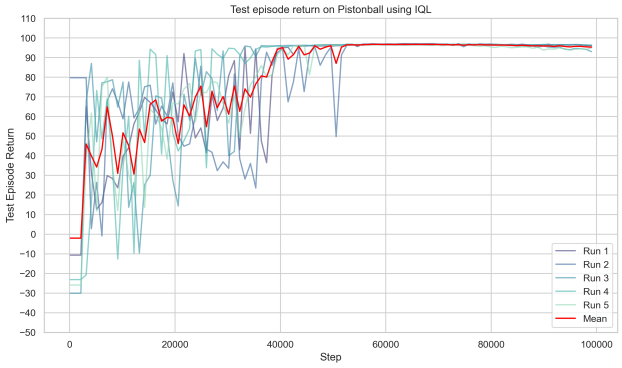


Figure 19: Pistonball discrete - IQL

Regarding the hyperparameters search, the ones we changed compared to QMIX and VDN is that we had to decrease the ϵ in 70,000 steps, needing more exploration. Besides, we had to increase the network size, using 256 units per layer instead of 64.

Looking at the results, we achieve optimal performance in all runs in the same steps as QMIX and faster than VDN. IQL needs more steps to reach a return of 80 but then suddenly raises to the optimal. VDN gets to 80 faster but takes more time to achieve the optimal, and QMIX gets to 90 a lot faster but takes time to achieve the optimal. Besides, IQL does not present those small oscillations the other two show at the final stages of training. The only concern is the run that starts falling at the end.

IQL learns the optimal behaviour even when decentralizing training because the agents do not need to coordinate to solve an episode. We need one piston starting the movement and the others keeping it until the end. Nonetheless, after simulating the testing episodes, we have seen how some coordination helps solve them faster overall.

The pistons only need to lower down when they do not detect the ball and raise when it is over them to make it move. The ball always starts at the rightmost pistons, so they usually raise while the others lower, starting the ball movement. As the ball advances, the others raise to keep it moving until the end.

Moreover, each piston observes its neighbours to know their state to decide its action. As that piston does not need to know about further pistons, it gives another reason why we can train them decentralized and still learn optimally. Finally, the discrete actions also facilitate IQL learning the optimal behaviour because a single one can move the ball further than a continuous one, needing fewer pistons to help.

However, IQL learns slower than the other two methods, especially QMIX, because it ignores the other agents and trains decentralized. Hence, the network suffers more wrong updates because it uses Q_{tot} to compute the action, trying to increase the joint action values for actions he thinks lead to a high shared reward, but it was due to the other agents. IQL works in this environment because it is simple enough and does not need coordination. However, in a different one, it would probably fail.

We simulated the same testing episode as previous methods, and we show the frames in Figure 54. If we look at how it solves that episode, we can see how it starts similar to the one from QMIX. However,

as the pistons have not learned any coordination, each acts when the ball is over them. Some pistons raise more than needed, others lower when they should not, and others directly do not move, the ball moving slower to the end, taking 20 steps to complete the episode.

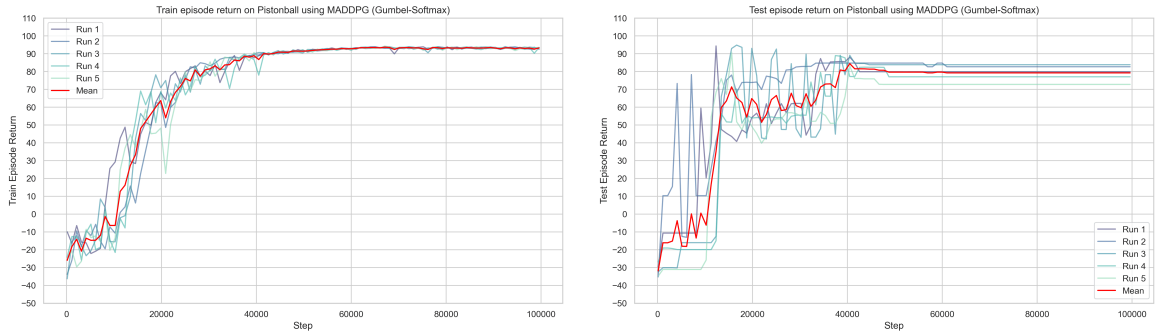
7.1.4 MADDPG

As explained, MADDPG can only work with continuous actions, but their authors applied the Gumbel-Softmax trick to use it with discrete ones. We leveraged EPyMARL's implementation, which uses TD(λ) to compute the targets when updating the centralized and monolithic critic. However, we did not observe a significant difference when using it or not.

The Gumbel-Softmax trick samples noise from the Gumbel distribution to modify the actor's logits. From those modified logits, it computes a softmax to obtain a probability distribution. Then, instead of sampling one action from that distribution, it selects the action with the highest probability. However, as we have modified the logits with the noise, we might select a different action when encountering the same observation, generating exploration.

We performed a first hyperparameter study using the inherent exploration of the Gumbel-Softmax trick. Nonetheless, we obtained bad results, only one or two runs in some cases, reaching near-optimal performance, the others not learning anything. For this reason, we decided to change the exploration mechanism and use the same ϵ-greedy exploration we have used in the other methods, maintaining the training loop identical (i.e. the loss functions).

We repeated the executions with this other exploration mechanism, which worked better but also had a problem. In Figure 20, we show the results of the best configuration, presenting both training and testing plots. While the training performance keeps improving until reaching an episode return of 93, stable across the five runs, the testing results show a flat line in all runs, not a single one achieving a return of 90, and all plateauing at different points.



(a) Training episodes

(b) Testing episodes

Figure 20: Pistonball discrete - MADDPG (Gumbel-Softmax trick)

All the other methods using ϵ-greedy presented more-or-less similar curves between training and testing, especially at the end. The difference is always higher at the beginning, with higher oscillations in the testing episodes, but as training proceeds, the difference becomes lower or nonexistent. It makes sense because we decrease ϵ from 1 to 0.05, reducing exploration as training advances but never making it disappear.

However, when evaluating the testing episodes while training, ϵ is always 0. Hence, in the beginning, while the action-value networks or the policy network is still learning while exploring, executing the greedy action will produce bad results as we do not know the real greedy action at a particular

observation. However, MADDPG does not seem to follow the same pattern.

We achieved those results by decreasing ϵ in 70,000 steps and increasing the units in the centralized critic from 64 to 256. The testing results plateaued before the exploration ended, at 40,000 steps. Hence, we checked the actor gradient, and it was quite small but not zero, which means the network was performing small changes that probably could not change the distribution enough to change the greedy action. To confirm that, we checked the logits the actor was outputting after training ended, and we saw a big difference between the two discrete actions.

Even though we changed the exploration, we still used the Gumbel-Softmax trick when updating the policy network. Hence, even when the ϵ was still high, and we could explore both actions in one observation, the probability distribution assigned too much weight to the wrong action and could not change to output the good one. MADDPG authors could use the trick with the environments they tried. Nonetheless, DDPG update equations assume continuous actions and should not work with discrete ones.

As a final test, we simulated the testing episodes like in previous methods, setting the ϵ to the minimum value of 0.05 instead of 0. We thought that maybe, with the small noise and only two actions per agent, we could achieve the same results with the testing episodes, and we were right. All the runs achieved an average return of 93 in the 25 testing episodes we ran.

The learned policies could still complete most episodes when executing the greedy action. However, they could not finish some without running random actions occasionally. For example, there was an episode where the ball stopped, and no piston was moving (i.e. being at minimum/maximum height and trying to descend/lift). Thanks to the piston under the ball randomly raising at some point, the ball started moving, and the other pistons reacted to help it reach the end. Without that random action, the ball remained stopped until the episode ended.

Those results confirmed our theory that the method learned to output the wrong greedy action in some observations, leading to worse performance without the random action. It is true that with the original exploration mechanism, we did not encounter that problem, and some runs could surpass a return of 90 in the testing episodes. However, the overall results in all runs were much worse than using ϵ-greedy exploration, so we preferred to keep it even if we suffered that other problem.

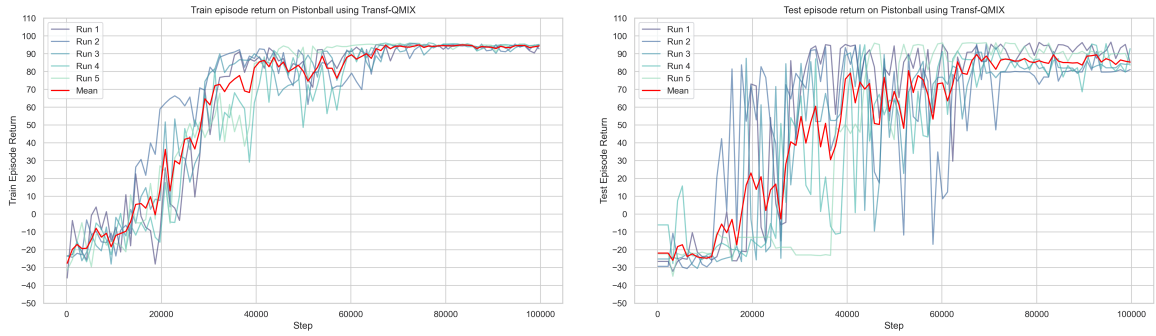
We tried training the method again but setting the minimum ϵ to 0 when training to see if we could learn the same behaviour without sometimes executing random actions, which could be very dangerous in some tasks. Nonetheless, even if we spent all training minimizing the ϵ, we could not achieve the same results in testing. Setting it to 0 earlier resulted in zero-gradient curves because we removed exploration, which we should always keep some as we did in all methods.

Finally, we present the frames of the first testing episode in Figure 56. Without executing random actions, the best policy could solve it in 32 steps. We can see how the rightmost piston tries to raise, but its left neighbour lowers, wasting its efforts. The ball starts moving because that left neighbour raises, slowly moving until the fifth rightmost lifts to speed up the ball, which is the last one helping it reach the end. The others start raising after the ball has crossed, not helping move the ball.

In the frames we present, it always seems that the pistons move after the ball has already crossed. It happens because of the simulation, but the piston helps push the ball forward. Nonetheless, in this case, the pistons do not help, but they have learned to lift when the ball is not around, as we can see in the rightmost ones that still move when the ball is almost at the other end.

7.1.5 TransfQMIX

We used the original TransfQMIX implementation [8] because it follows the same structure as EPyMARL. As explained, the method leverages transformers to learn the agent networks and the mixer and is the only method we executed using such network architecture. In Figure 21, we show the best results we could obtain:



(a) Training episodes

(b) Testing episodes

Figure 21: Pistonball discrete - TransfQMIX

We plotted the training and testing results because the training again achieved higher return and stability. As the method uses ϵ-greedy exploration, we thought the problem could be the one we encountered before of the network learning to output the wrong action for some observation, as we only have two. We were right, and once we set the ϵ to 0.05 in the testing episodes, we achieved the same results as the training plot.

To train the method, we had to decide how to create the observation and state matrices, deciding how many entities each agent observes, and the set of features in each matrix. To generate the observation matrix, each agent only observes itself as an entity, the set of features being the four flags we compute. Even though we use information about the ball and the neighbours, we do not consider it as features of other entities but of the current piston.

We did not leverage the features the authors designed because we did not need agent identifiers. Moreover, as the agents only observe themselves, we do not need to indicate if the features belong to an agent entity. To generate the state matrix, each agent observes itself, the only feature being the single flag indicating if the ball is over itself.

In terms of hyperparameters, we obtained those results by setting the default values for the transformer-related ones. We used the same values in the shared agent network and the mixer. In particular, we used four parallel heads, two transformer blocks, and an embedding dimension of 32. For the other parameters, we decreased ϵ in 50,000 steps, used a learning rate of 0.0005, a gradient clipping of 500, and TD(λ) to compute the targets to update the networks.

We simulated the first testing episode with the best policy TransfQMIX learned to see how it solved that episode, and we present the frames in Figure 55. It took 23 steps, and TransfQMIX solved it similarly to MADDPG, but faster. The pistons learned to lift way before the ball was almost over its left neighbour, unlike MADDPG, which generated higher momentum for the ball to move faster.

Nonetheless, looking at the results the authors reported in the environments they trained, we did not understand why we obtained such results, even though we used the same implementation. In all

[8] https://github.com/mttga/pymarl_transformers/tree/main

experiments we performed, we observed the same problem of the training plot displaying better results.

We spent a decent amount of time with this method, but due to our shortage of knowledge in transformers, we might have overlooked something, explaining why we could not obtain better results. As we had a lot more to do, we decided to leave it as future work and spend the needed time to see if we could get results comparable to the ones reported.

7.1.6 DQN

DQN is a single-agent method, meaning a single network needs to control all pistons and output one action for each. The network receives the fully observable state as input and learns to approximate the joint action-value function. Nonetheless, it differs from the shared network we trained in MARL methods. Here, the network must learn to map the state to the joint action values of all the possible actions all pistons could execute. Thus, it has to output $2^{number\_of\_pistons}$ action values.

To select the action to execute at the environment at each step, it has to compute an *argmax* to find the greedy action and decide which one to execute using ϵ-greedy exploration. For this reason, we decided to remove the action that does not move the piston. With 15 pistons and two actions per piston, it outputs 32,768 action values instead of the 14 million it would output with three actions.

We performed a first hyperparameter study with 15 pistons, but the results were awful in all configurations. In all cases, the five runs did not learn anything useful and could not solve any episode. Thus, we reduced the number of pistons to 10 and performed the hyperparameter study again to ensure we implemented the method correctly and could solve an easier task. In Figure 22, we show the best results we obtained:

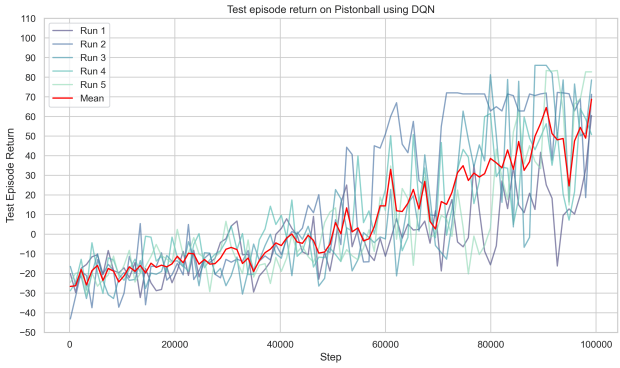


Figure 22: Pistonball discrete - 10 pistons - DQN

After studying the hyperparameters with ten pistons, we found two worth mentioning. First, like previous methods, we obtained better results when increasing the number of units per layer from 64 to 256. Second, we discovered that standardizing the rewards and the returns was essential for DQN to learn something, even with ten pistons.

We took the idea from the EPyMARL implementation of QMIX, which allowed the user to standardize the rewards and the returns to avoid problems with exploding gradients. They did not implement DQN, but we thought we would try it anyway, considering anything we tried before applying this trick did not work with ten pistons.

The performance across the different seeds does not reach an episodic return of 80 if we consider the mean line. It learns slowly but is understandable considering how the network needs to explore all the

1024 options compared to the multi-agent methods where a network only explores two. Nonetheless, we expected better results, considering how we reduced the task complexity compared to the MARL methods.

We could probably obtain better results if we allowed the method to train for more steps and explore the action space more, as the average performance seems to keep improving. Nonetheless, ten pistons could already be too much for DQN to handle, never reaching optimal performance. In any case, it already works worse than MARL methods trained with more pistons.

Even with the low performance with ten pistons, we experimented again with fifteen to see if we could improve the first results, considering we had to compare them anyway with the MARL methods. Even with the trick of standardizing the rewards and returns, it did not work with anything we tried. In Figure 23, we show the results of the same configuration we used to obtain the results of Figure 22, but with fifteen pistons:

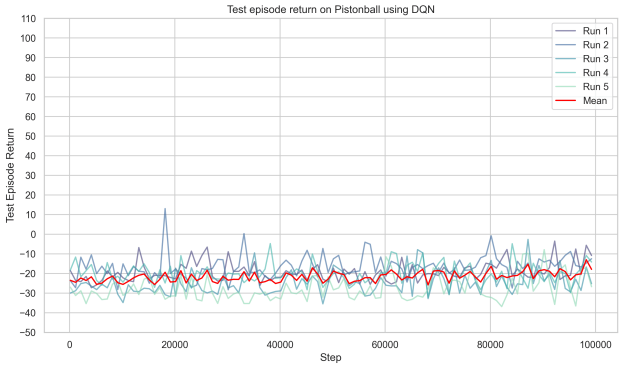


Figure 23: Pistonball discrete - 15 pistons - DQN

The method failed to learn anything again in any of the five runs, not learning to complete any episode. At each step, it has to select an action for each piston among 32,768 options, which requires more exploration than what 100,000 training steps can offer. Besides, even with enough training time, we believe the network could not learn to correctly approximate the joint action-value function such that the greedy action would be optimal for each piston.

Finally, we wanted to simulate the testing episodes when using ten pistons to see what DQN learned using the weights of the best run. However, we did not include the frames because we could not compare them with previous methods, and nothing was worth seeing. Again, DQN suffered from the same problem as MADDPG, of executing random actions occasionally to solve certain episodes where the greedy action could not take the ball to the end.

We mostly observed the pistons reacting when the ball was over them, lifting to move it. However, there was not any coordinated strategy like previous MARL methods. Besides, we observed a more random behaviour from the pistons, mostly moving for no reason and hindering the ball movement in some cases instead of helping it.

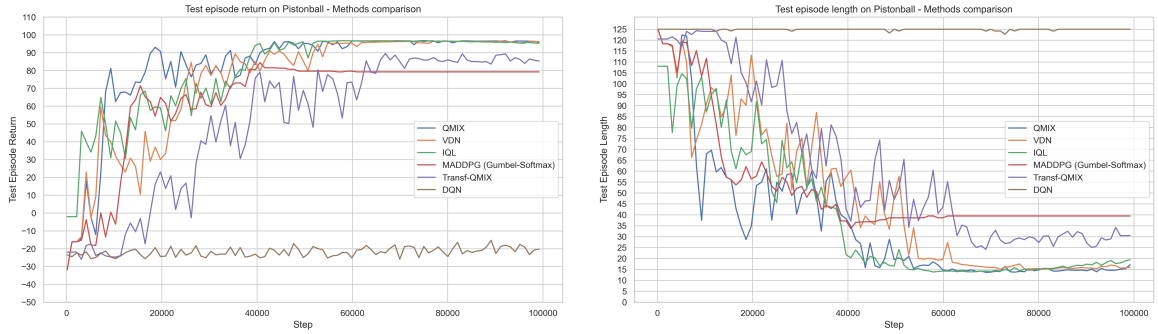
This behaviour can happen because DQN needs to output the action for each piston that maximizes the joint action value. Hence, it might output a good one only for a subset of pistons, not caring which action outputs for the rest. Ideally, it would output the optimal one for all. However, as the method did not explore the action space enough, it did not have time to learn the best action for all the pistons.

7.1.7 Comparison

Let's compare the best results we obtained with each method. For this reason, we will plot each method's average testing return (i.e. the red line). However, as one episode can end earlier, and the best policy should solve it the fastest, we will generate a second plot comparing the average testing episode length.

When training each method, we generated a second testing plot where we plotted the average episode length instead of the average episode return of the 25 testing episodes we executed each certain number of training steps. We generated it identically, so we do not explain it again.

In Figure 24, we show the two plots comparing the different methods. Nonetheless, we have not applied the trick of suddenly executing random actions in MADDPG, DQN, and TransfQMIX.



(a) Episode return

(b) Episode length

Figure 24: Pistonball discrete - Methods comparison

QMIX, VDN, and IQL learn the optimal policy in all five runs in 60,000 steps, with QMIX learning the fastest, followed by IQL. If we look at the episode length, we can see how the three methods are similar once they learn the optimal policy. However, in the end, IQL takes, on average, five more steps even though the return does not significantly change because of that run decaying.

QMIX is the best of those three because it does not suffer from a strict decomposition or the joint reward's non-stationarity. Thanks to learning a mixing network, it can adapt the factorization as training proceeds to focus first on some agent that might contribute more. Moreover, thanks to training a shared network without identifiers, focusing first on learning that agent's behaviour helps learn the others faster, learning the optimal policy the fastest.

VDN should learn faster than IQL because it leverages centralized training. However, the difference is small and falls behind at some point. VDN decomposes Q_{tot} into a summation, which can be too strict and slow down learning at the first stages, especially if the number of agents is high. Decomposing Q_{tot} into a summation causes the gradient to more-or-less equally divide into all agents, performing small network updates. Moreover, as we trained a shared network, it is easier to wrongly update the current approximation because of some agents' bad actions.

IQL does not need to worry about one agent contributing more because it ignores the other agents and trains each decentralized to learn to approximate the joint action-value function. Nonetheless, it learns slower than QMIX because it decides the actions based on Q_{tot}, which still depends on all agents' actions. Thus, sometimes wrong updates happen because the network thinks an action achieves a high return, but it was due to another agent's action.

Next, we have TransfQMIX, for which we could not obtain better results. It still achieves very high

results, but the cost of training the transformer is not worth it. We are sure it can achieve optimal results, but we could not get through it to understand what was happening. Thus, we left it as future work.

The fourth method would be MADDPG, which presents a good learning pace until the 40,000 steps when it suddenly flattens in all five runs. We believe the problem is adapting MADDPG to discrete actions by leveraging the Gumbel-Softmax trick but still keeping the continuous update equations.

Even though we changed the exploration mechanism, it was useless because once the policy shifted to one from which it could not change anymore, the gradients became very small. And even if exploration remained until the end, the greedy actions were not always optimal, leading to sub-optimal testing performance. For this reason, suddenly executing the non-greedy action led to higher results.

Finally, we have DQN, which could not learn to solve any episode with 15 pistons. With 10, it could solve them sub-optimally because the number of action combinations it had to explore was feasible. However, when increasing to 15, it was impossible to explore enough of the action space in those few steps.

We can conclude this comparison by saying that leveraging MARL methods to solve the task was essential. Moreover, those methods factorizing the joint action-value function worked better than the one we executed that did not perform such decomposition, even if we applied the trick of keeping the exploration in the testing episodes. Finally, at least one factorizing the joint function worked significantly better than the one training decentralized.

We executed MADDPG as the representative of methods that do not decompose the joint action values. One could ask why we did not try LICA, which the authors affirmed worked better. Nonetheless, we found it way after we finished training all methods in this environment, given it was the first we experimented with. We did not have time because we were always busy with some experiments, so we left it as future work.

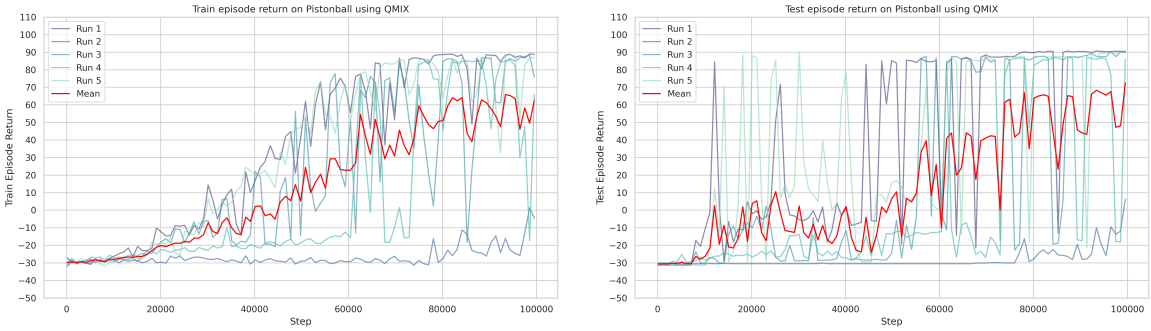
7.1.8 Scalability

We wanted to increase the number of pistons considerably and demonstrate how we could reuse the policy learned by QMIX (the one with the best results) and still achieve very high results without any extra training. We can perform this test because all the pistons behaved identically, so we trained a shared policy (or action-value network). It would be impossible if we had trained a network per agent.

With 15 pistons, the maximum return per episode was around 96, with the time penalty we used. If we increase the number to 100, the maximum return is around 92 because the ball needs to travel further. First, we executed the testing episodes with 100 pistons using one of the QMIX's learned policies. Without training it further, we obtained an average return of 90 and an average episode length of 40.

We rendered the first one to see how QMIX solved it, and we saw the ball a few times going back. We also saw the ball spent some unnecessary steps over some pistons because they were blocking its movement by lifting and descending repeatedly. Moreover, the ball stopped twice at the middle pistons, and they had to raise to start moving the ball again. Nonetheless, QMIX solved the episode in 38 steps.

Next, we trained from scratch QMIX with 100 pistons, repeating the same process we did with 15, trying different hyperparameter combinations. We present the best results we could obtain in Figure 25:



(a) Training episodes

(b) Testing episodes

Figure 25: Pistonball discrete - 100 pistons - QMIX - Training from scratch

We can see how the maximum return some of the runs achieve is 90, the same as we achieved by executing the behaviour learned with 15 pistons. Moreover, the learning is very unstable, and the performance in the testing episodes oscillates a lot because the greedy action constantly changes. We also have one run that starts learning at the end, and the learning pace of the others is slower than when we trained with 15 pistons.

We tried the same hyperparameters we tried when training QMIX with 15 pistons, allowing higher and lower learning rates, exploration rates, and different gradient clipping values. To obtain those results, we had to decrease ϵ in 70,000 steps instead of 30,000 and use a smaller gradient clipping value (0.5 instead of 50).

We executed the testing episodes with the best policy QMIX learned with those 100 pistons, achieving an average return of 90 and episode length of 39 in the 25 episodes. Moreover, we rendered the first one to see how QMIX learned to solve it. QMIX solved it in 39 steps because the ball also went backwards. However, even if the ball did not stop, and some pistons did not hinder its movement, it needed one extra step because the pistons moved worse, and the ball moved slower.

Finally, we wanted to see if we could train QMIX with 100 pistons, but instead of training from scratch, start from the weights learned with 15 pistons and fine-tune them with the extra pistons. However, we had to discard the mixer weights, as they depended on the number of agents.

We tried a lower initial ϵ because we should not explore the action space that much but perform a few adjustments to some actions because the initial performance was already very high. Thus, we tested with 1, 0.5, and 0.05, the last one meaning we are at minimum exploration during all training. Besides, as we are reusing weights and should avoid modifying them too much, we also tried a lower learning rate, even though we already tried it in all previous executions.

In Figure 26, we present the best results we could obtain. We do not include the training plot, as it is almost identical as the testing one.

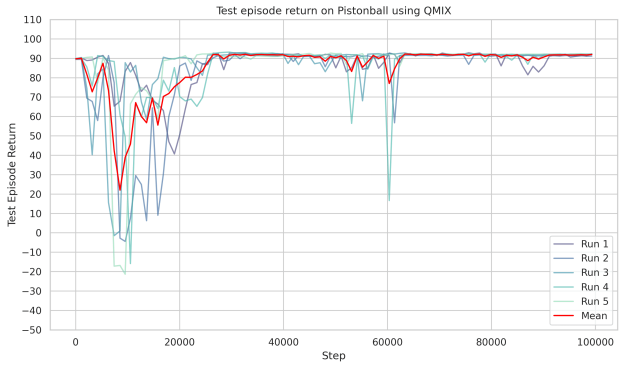


Figure 26: Pistonball discrete - 100 pistons - QMIX - Fine-tuning

We obtained these results by setting the initial ϵ to 0.05, meaning it was already on its minimum value. Moreover, we used a lower learning rate than in all previous executions with QMIX, 0.005, and a gradient clipping of 0.5. Lowering those parameters avoids modifying the action-value network initial weights too much. The performance still decays at the beginning of training because it has to train the mixer from scratch.

Nonetheless, thanks to the agent network already approximating the individual action-value function very precisely, in less than 30,000 steps of updating the mixer and the shared agent network, QMIX has adjusted the decomposition to approximate the joint action-value function for the 100 pistons, maximizing the joint return. It has learned the optimal policy in all runs faster than training from scratch or with 15 pistons. It presents some decay mid-training that later restores.

We executed again the testing episodes with the fine-tuned policy and achieved an average return of 92 and an episode length of 32, the best of the three. We also rendered the first one with one of the policies, and QMIX solved it in 28 steps. It did not present those unnecessary steps of the ball going backwards or some pistons hindering its movement. Moreover, the pistons moved more coordinated because the ball did not stop, and it moved faster.

We want to mention that we have not included any of the renderings of the testing episodes because, with 100 pistons, it was a little hard to see it with frames. Thus, we recommend looking at the GIFs we uploaded to our GitHub repository.

Therefore, if we have to solve a task where the agents behave identically, we can train a MARL method on a smaller version first (i.e. lowering the action dimensionality) and then reuse the learned weights and fine-tune for fewer steps than training from scratch the more difficult task, reaching the same policy. On the other hand, a single-agent method would fail to learn the more difficult task, as we have seen how DQN already fails with 15 pistons.

7.2 Pistonball Continuous

This environment is the same as the previous one, but now the actions are continuous, each sub-action taking values in the $[-1, 1]$ interval. With this action type, we did not give up on any action as we did in the discrete case because of DQN. Moreover, each sub-action being an interval gives a more fine-grained control of how to raise and lower the piston. Thus, we believed the continuous methods should achieve better results, especially in learning pace, given the discrete ones already achieved optimal performance.

Nothing else changed, so we still believed the pistons should behave identically, even if we changed

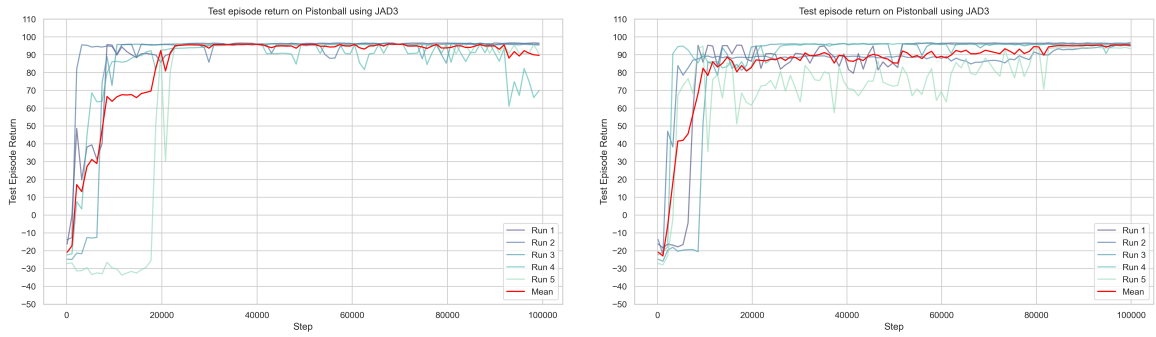
the action type. Considering how we already demonstrated that identical behaviour, we did not train again some methods with the three different network architectures of sharing all pistons, the middle pistons, or training one network per agent, and we always trained a single shared network, or rather, a shared actor and critic, considering how all the methods are actor-critic.

When experimenting with all the methods, we observed that, unlike the discrete case, adding the previous action with the observation as input to the networks did not help. None of the continuous methods use recurrent layers, and probably, the linear layers do not find any benefit from using a single action. The recurrent one could probably store certain information from multiple previous transitions, and the action was beneficial.

7.2.1 JAD3

With our new method, we wanted to test if it was better to update the actor using the joint action-value function approximation or use the individual ones learned through factorization. For this reason, we performed the hyperparameter study twice, changing how we updated the actor to compare which option was better.

In Figure 27, we show the best results we obtained when updating the actor with the individual or joint action-value function:



(a) Updating the actor with $[Q_a]_{a=1}^N$

(b) Updating the actor with Q_{tot}

Figure 27: Pistonball continuous - JAD3

Updating the actor with the individual function seems to work better than using the joint one, even though we have a run in the former that suddenly decays. We expected the former to work better than the latter, although we were not sure because we trained a shared network without identifiers due to all agents behaving identically.

Nonetheless, even though the agents behave identically, they do not have the same importance in solving the task. Hence, updating the action-value network using the individual functions is easier than with the joint action value, which can confuse the network and cause wrong updates until it deciphers how each agent needs to act from that shared scalar value.

When updating the actor with the individual action values, all runs achieve optimal performance in less than 25,000 steps, even though some oscillates, and we have that run suddenly decaying. However, we looked at the training results, and that same run was improving again. Thus, if we trained it for more steps, it would learn again the optimal policy.

On the other hand, updating the actor with the joint function learns a good policy in all runs faster, achieving a return of 90, but taking a lot more steps for all to learn the optimal policy due to three runs

oscillating considerably during most training. Nevertheless, after 80,000 steps, all the runs are stable at optimal performance.

We studied the same hyperparameters for each updating mechanism and found almost the same best values for both. There are four worth mentioning. First, increasing the size of the mixer network's layers from 32 to 64 was essential, which is acceptable as we have to merge 15 values.

Second, we increased the gradient clipping value from 0.5 to 500. It is possible that the fourth run in the first plot suddenly decays because we allowed such a bigger gradient, considering how that run's performance oscillates all the time. Due to all agents sharing the network, a wrong update might have happened, breaking down the policy. Nonetheless, it did not happen when updating the actor with Q_{tot}, and either way, a smaller value gave us worse results.

Third, we increased the noise clipping value in the target actions from 0.5 to 1 when updating the actor with the individual functions. Again, that value may be too high and also be responsible for that run decaying. However, again, using a smaller value led to worse results. In any case, training for more steps would restore the optimal policy, but we decided not to re-execute everything again for more steps.

Fourth, we found that a small σ in the Gaussian noise we added to the exploratory and target actions was needed, especially if the gradient was that high. We used 0.2 as the starting value and decreased to 0.05 during the 100,000 steps. Using a bigger one caused the actions to grow to the maximum values, the policy shifting to output them all the time, and the gradient becoming zero.

As we did in the discrete case, we simulated the testing episodes using the policies we learned when updating the actor with $[Q_a]_{a=1}^N$ and Q_{tot} to see if there was some difference in how they solved the task. We tried the policies of each run in both cases. Those learned with Q_{tot} did it faster, on average. Nonetheless, we saw that both learned similar strategies.

In Figure 57, we show the frames of the first testing episode solved by the best policy learned when updating the actor with $[Q_a]_{a=1}^N$. The two rightmost pistons raise for some steps while the others lower to their minimum height. With that height difference, when the ball falls, it has enough momentum to move far and fast, the others raising a little to ensure it reaches the end. Thus, as the others still help move the ball, spending that many steps lifting those might be inefficient. Nonetheless, thanks to the others being at their minimum height, they also avoid stopping the ball due to some collision.

In Figure 58, we show the same testing episode but solved by the best policy learned when updating the actor with Q_{tot}. It solved the episode in 14 steps compared to Figure 57, which needed 18. The two rightmost pistons also lift for some steps while the others lower, but they spend fewer steps, and the ball starts moving earlier, the other pistons raising a little when the ball passes to keep it moving until the end.

Even though the policies learned when updating the actor with Q_{tot} solved the episodes a bit faster, we still consider the ones trained with the other update mechanism optimal. The difference in steps is low if we assume the best policies, usually two or three extra, and in some episodes, it needs the same or fewer than the other. Moreover, the one updating the actor with $[Q_a]_{a=1}^N$ learned them faster. Hence, we would still consider a better option to use the individual action-value functions in the actor loss computation.

7.2.2 FACMAC

FACMAC authors did not execute this environment in their experimentation, so we had no results we could compare with. Nonetheless, we implemented FACMAC from scratch instead of using their code because it was a little messy to understand and integrate with ours. Moreover, we performed a

small modification, decreasing the Gaussian's σ as training proceeds to a minimum value, like our new method, instead of keeping it fixed like the original implementation.

We performed the usual hyperparameter study, but we could only achieve optimal performance in one run near the end. In Figure 28, we show the best results we could obtain:

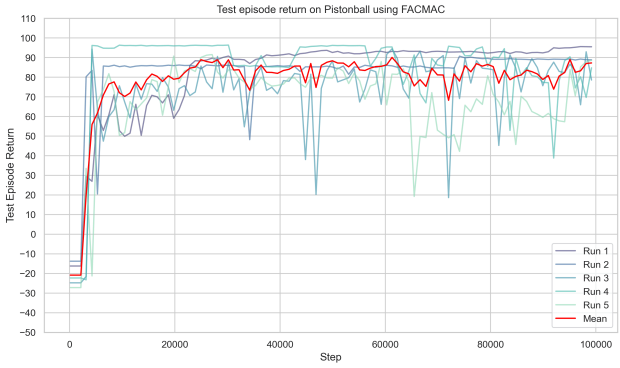


Figure 28: Pistonball continuous - FACMAC

As we saw in the previous method, setting a low initial value for the Gaussian's σ we decrease during training was important. Setting it to 0.2 gave the best results again. Second, we also increased the gradient clipping value, but now we found 50 instead of 500 to work better. Finally, we also tried exploring the size of the mixing network, but using 64 units per layer worked worse than 32. We assume the parameters change compared to our new method because FACMAC suffers from overestimation. After all, it lacks TD3's improvements.

If we look at the results, one run learns the optimal policy pretty fast but starts oscillating, and in the end, it becomes sub-optimal. There is one that learns slower than the rest but keeps improving steadily until becoming optimal at the end. And with the other three runs, we have big oscillations that never settle down. Overall, the mean line is stable between 80 and 90, but it never improves because of those oscillations and that first run becoming sub-optimal.

Our new method when using Q_{tot} to update the actor is FACMAC but adding TD3's improvements. We saw how it achieved optimal performance in all runs at the end without presenting those high oscillations. The hyperparameters are not the same as those in our new method. However, we tried them also, and FACMAC obtained worse results. For this reason, FACMAC works worse because it suffers from overestimation problems the TD3 improvements mitigate.

We simulated the first testing episode with the best policy, but we decided not to include it here because it behaves like Figure 58, but worse. The episode starts similarly with the two rightmost pistons raising and the others descending. However, when the ball starts moving in the fifth frame, the next pistons have not lowered enough, and the height difference is not enough, stopping the ball. Thus, those pistons descend to start moving the ball again, the others raising as usual to keep the ball rolling. Moreover, the ball moves slower than in Figure 58, taking thirty steps to complete.

7.2.3 IQL

We created the continuous version of IQL by using FACMAC as the basis and removing the mixing network. The name should probably be different, but we kept the same to facilitate the explanation. We can train an actor and critic per agent, each actor learning to approximate the agent's policy and each critic, the joint action-value function. Each agent assumes the others do not exist. Thus, each uses

its information to train both networks to maximize the joint action-value function.

Nonetheless, as all the agents behave identically, we can train a shared actor and critic, like the discrete case. However, even though all agents share networks, they still learn decentralized using their information but not from the other agents.

In Figure 29, we present the best results we have obtained, and compared to the discrete version, IQL could not achieve optimal performance in all five runs, only one near the end. Moreover, most runs spent around 60,000 steps without improving, one even becoming flat because of a zero gradient, and the others never reaching a return of 90.

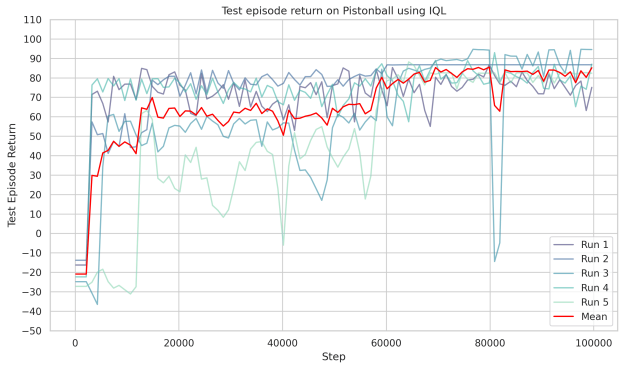


Figure 29: Pistonball continuous - IQL

We performed the usual hyperparameter study, and like previous methods, we found that using a σ of 0.2 was the best option to explore without pushing the actions to the maximum values, especially when using a gradient clipping of 50, which again worked the best. However, unlike the discrete case, increasing the number of units per layer did not improve anything, so we kept the default value of 64.

IQL achieves similar results to FACMAC's, especially at the end. FACMAC learned policies that could surpass a return of 80 in fewer steps, but the performance worsened. The difference is that IQL does not have the mixer to centralize training, but the critics learn Q_{tot}. Even if FACMAC does not leverage the learned factorization, it should work better thanks to using all the agent's information plus the fully observable state. We believe IQL achieves similar results because of the environment's low complexity and need for coordinated behaviour.

Compared to the results IQL achieved in the discrete environment, it works worse in the continuous version because the more fine-grained actions require some coordination between the agents. In the discrete case, a single action would lift or descend more the piston, the ball gaining more momentum to move further and requiring less help from other pistons.

On the other hand, in the continuous case, the actor will never learn to output actions (without adding noise) with maximal value as it would mean the *tanh* being at its flat zone, the gradient becoming zero, and the policy not improving anymore. Hence, to move the ball like in the discrete case, a piston will need the help of another or raise for several steps without the ball moving, which is difficult to happen except for the two rightmost at the beginning.

To understand the difference with discrete IQL, we simulated the testing episodes with the best policy to see how the continuous IQL solved them. Moreover, we also compared them with the previous continuous methods. Overall, continuous IQL worked worse than previous continuous methods and discrete IQL. In Figure 59, we present the frames of the first testing episode, which took 39 steps to

complete.

Discrete IQL solved it in 21 steps, with the ball moving faster and the pistons reacting when they detect the ball, lifting to make it move or keep it moving. Besides, the pistons randomly moved when the ball was not around because they lacked the action to remain still. In the continuous case, the pistons solved the task essentially the same, the ball moving slower because the pistons lift less. Hence, they do not act coordinated as in our new method.

Each piston has learned to maximize Q_{tot}, always performing an action that would give a high shared reward, like lifting the moment they detect the ball. They cannot sacrifice for the greater good because they do not consider the other agents. Hence, when the episode starts, the rightmost pistons move to start the ball movement. However, that movement is slow, and the pistons cannot speed it up with a single action. Hence, they can only lift when detecting the ball to keep it rolling, explaining why the episode needs more steps than previous continuous methods.

7.2.4 MADDPG

We implemented MADDPG from scratch, following the implementation the authors of FACAMC used in their experimentation to compare with FACMAC. Compared to the discrete version, the continuous one trains an actor to directly output actions instead of logits, using a *tanh* activation. Besides, it needs some exploration mechanism, as it learns a deterministic policy. Like we did with the previous methods, we added Gaussian noise.

However, like with the discrete environment, MADDPG got the worst testing results. In Figure 30, we show the best we could obtain, which is not bad in some runs, but what is surprising is how, in four, the gradient in the actor becomes zero, and the policy stops improving, the testing results not changing either. Unlike the discrete case, the training results are equally bad.

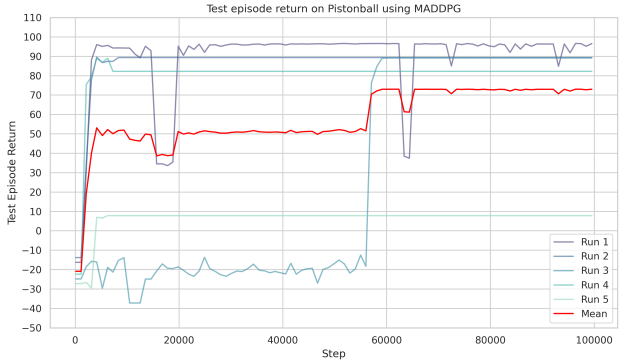


Figure 30: Pistonball continuous - MADDPG

When performing the hyperparameter study, we observed at least two or three runs per experiment with a flat line because the actor's gradients became zero. We obtained the best results by setting the gradient clipping value to 50. Nonetheless, we had to decrease the initial Gaussian σ to 0.1 because the one used in previous methods led to worse results. Finally, like the discrete version, increasing the number of units per layer in the centralized critic to 256 worked better.

We also tried several tricks to avoid the gradients becoming zero so fast. For example, all continuous methods sample random actions for some steps at the beginning to fill the replay buffer instead of using the ones the policy outputs. We tried increasing considerably compared to previous methods as it would increase the exploration, the policy seeing more different actions. Nonetheless, it was futile. We also

tried adding previous transitions with the current observation as input to the actor, but it did not work either.

MADDPG could more or less solve the discrete task in all runs, but when switching to continuous actions, it cannot. MADDPG struggles because the observations and states are flags. Thus, it does not have a lot of possibilities to visit to try actions, not exploring the action space enough before the actor converges to some sub-optimal policy. With the discrete task, it had fewer problems because there were only two actions per agent, and ϵ-greedy ensured visiting all the possibilities.

We simulated the testing episodes with the best policy to see how MADDPG solved the task. When averaging the number of steps to complete the 25 episodes, MADDPG achieved a slightly better average than our new method when updating the actor with $[Q_a]_{a=1}^{N}$ (i.e. 15.44 vs 15.72), but worse than when updating it with Q_{tot} (i.e. 14.8). For example, it solved the first testing episode exactly like Figure 58, so we do not include its frames again. Of course, we still consider MADDPG worse because only one run is on the same level as our new method.

7.2.5 TD3

We implemented TD3 from scratch, following the original implementation from [13]. Like DQN, TD3 trains a single actor to approximate the joint policy using the fully observable state as input, even though the environment is initially multi-agent. The network learns to map the input state to as many actions as pistons, fifteen in this case. It also trains a single critic to approximate from the same state the joint action-value function, given the environment has a shared reward.

Unlike DQN, we directly tried to solve the environment with 15 pistons because training TD3 took less time as it did not compute an exhaustive maximum. However, even though we had the full action spectrum and more fine-grained control of the pistons, TD3 could not solve it with that many pistons, not even learning anything. In Figure 31, we present the best results we could obtain:

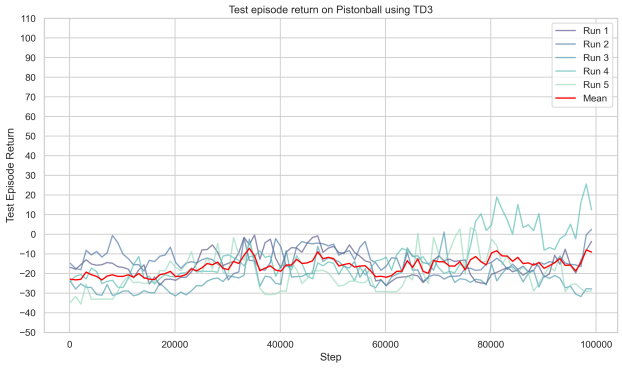


Figure 31: Pistonball continuous - TD3

The five runs display negative or zero performance, with three starting to improve near the end. Thus, the problem is learning to output 15 actions with a single network, not exploring the joint action space enough with only 100,000 steps.

MADDPG does not learn optimally in all runs, but thanks to the actor learning to control a single piston and only exploring one sub-action, it learns a decent policy in very few steps. The centralized critic in MADDPG is similar to the one in TD3 but also receives the actions. However, learning individual policies instead of the joint one makes the real difference.

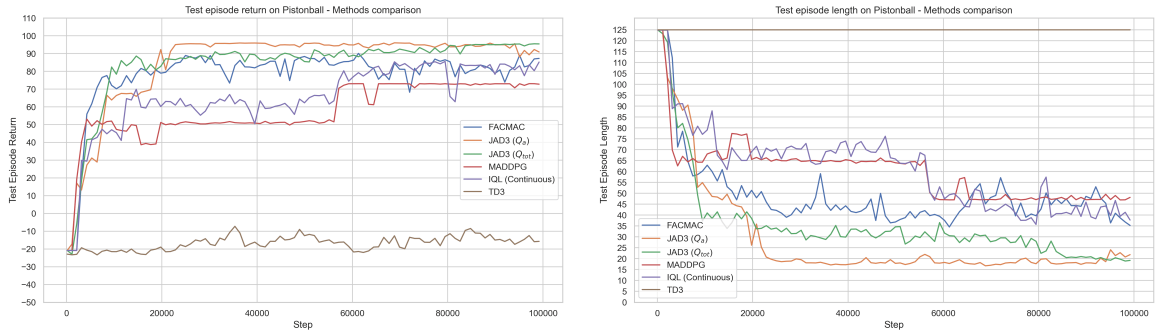
We performed a hyperparameter study, changing all the possible parameters TD3 has, but it was futile, and no parameter could improve what we have presented above. We tried using a higher value than usual to clip the gradients, considering how the original TD3 did not clip them.

We also tried standardizing the returns and the rewards as we did in DQN. However, we did not expect this trick to work, considering how the original TD3 implementation did not require this trick to learn optimal policies in environments where the maximum return was higher. Of course, it did not work either in this environment.

We simulated the testing episodes as usual, but no run could complete one and all episodes finished before the ball could reach the end. It is not that the ball could not even start moving, but the pistons could not learn to coordinate their movement, not even react when the ball was under them. The ball always got stuck at some point, the pistons under it not moving, and others blocking its way.

7.2.6 Comparison

As we did in the discrete case, we have plotted together the mean testing results for each method (i.e. the red line) of the average return and the episode length. With JAD3, we have plotted the results obtained with each update mechanism. In Figure 32, we present the two plots:



(a) Episode return

(b) Episode length

Figure 32: Pistonball continuous - Methods comparison

Our new method achieves the best results independently of how we update the actor. All or almost all runs learn the optimal policy before training ends, and the performance remains stable. We indeed had that run that suddenly decays a lot, which worsens the mean performance when updating the actor with the individual action values. Nonetheless, we are sure it would recover and learn again the optimal policy with a few more training steps.

FACMAC is the second method in terms of results and the other decomposing the joint action-value function. The performance at the first stages of training was similar to JAD3 (Q_{tot}) but with higher oscillations that remained during all training. Nonetheless, instead of improving the results as training proceeded, they worsened. We expected it to work worse than JAD3 (Q_{tot}) because it lacks the TD3 improvements that should help mitigate the overestimation problems.

FACMAC's results ended similar to IQL's, but at least better than MADDPG. The pistons learned some coordination, but it was not enough. Even if it does not leverage the decomposition and updates the actor with Q_{tot}, training centralized using all the available information should give better results than IQL.

FACMAC uses the same mixer as QMIX, and considering how JAD3 learns optimally, the problem has to be the overestimation problems hindering the learning of that mixer. IQL should suffer from similar problems, plus the non-stationarity of learning to approximate the joint function. However, as it does not need to learn such decomposition, it suffered less, not showing such performance decay.

IQL's results are decent if we consider how it trains fully decentralized without leveraging information from the other agents. Indeed, this environment does not require a high degree of coordination, and each agent learning a policy to maximize Q_{tot} is enough to reach good performance. However, IQL solved the task sub-optimally because the training was decentralized, and the pistons did not learn cooperative behaviour.

With MADDPG, we were a little frustrated because we could not solve the problem of the zero-gradient curves, even though we tried a lot of parameters. The mean results are not bad because most runs surpass a return of 80, but only one could achieve more than 90. As with FACMAC, the pistons learned some coordinated behaviour, but not enough to achieve optimal performance.

Finally, we have TD3, the single-agent method. It did not learn any decent behaviour that could solve any episode with that many training steps because the network has to learn to control all 15 pistons. We did not bother to train it longer, as the other methods did not face the same problem.

We can conclude this experiment by saying that leveraging MARL methods could solve a task that a single-agent one could not. Moreover, we achieved the best results with those decomposing Q_{tot}, especially with JAD3, demonstrating how it is better to factorize the joint function and centralize training.

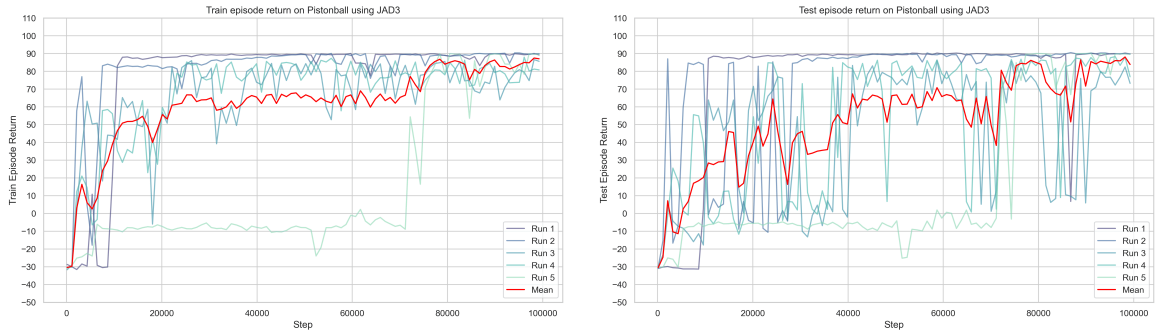
7.2.7 Scalability

As we did in the discrete version of the environment, we also increased the number of pistons to 100 to see how, if we fine-tuned the policies learned with 15, we could learn the optimal faster than training from scratch with those 100 pistons. Nonetheless, we can do that because the agents behave identically, and we have trained a shared policy for all.

We used the best policy we obtained with JAD3 when we updated the actor with $[Q_a]_{a=1}^{N}$ and started like in the discrete case by testing how the learned policy behaved when increasing the number of pistons to 100 without training it further. Like the discrete case, it achieved an average return of 90 in the 25 testing episodes, taking, on average, 40 steps.

We rendered the first one to see the behaviour. It solved it identically to what we saw in Figure 57. However, with 100 pistons, the ball takes more steps to reach the end because even if we lift the two rightmost ones for some steps while lowering the others to create momentum, the ball cannot get very far, and more pistons in the middle and left sections must help. Moreover, the ball stops once, as we saw in the discrete case.

Then, we decided to train the new method with those 100 pistons from scratch to see if it could learn the same or a better policy. However, 100,000 steps may be too few, as we saw in the discrete case. In Figure 33, we show the best results we could obtain, showing the training and testing plots:



(a) Training episodes

(b) Testing episodes

Figure 33: Pistonball continuous - 100 pistons - JAD3 - Training from scratch

Training from scratch with 100 pistons in 100,000 steps is difficult, even though it presents better results than the discrete case. We also have differences in the training and testing plots, presenting higher oscillations in the testing episodes because the action of the deterministic policy outputs changes. Nonetheless, the average results across the five runs always increase, reaching almost a return of 90 at the end.

We can see how no run could achieve the return of 92 we saw in the discrete case. We believe the difference is the continuous actions, not lifting or raising the pistons that much, which does not allow them to surpass 90. We can also see how some runs learned pretty fast the best policy, while others took more time, some presenting high oscillations during most training.

We got those results by using an initial σ of 0.2 in the exploratory actions, like when we trained with 15 pistons, but we decreased the one we used to generate noise in the target actions to 0.1. Moreover, we also lowered the learning rates and the gradient clipping in both actor and critic, using a value of 0.0005 in the learning rates and a clipping value of 0.5. Finally, we maintained the clipping of the target noise at 0.5 instead of 1.

We simulated the first testing episode again to see how it learned to solve the task. The two rightmost pistons did not lift for some steps but were their two left neighbours that lifted for a few steps, but less than what we saw before, to start moving the ball. At the same time, the remaining pistons descended to their minimum height, lifting when the ball was over them to keep it moving. Nonetheless, it solved the episode essentially the same, with the ball also stopping once.

Finally, we trained again the method with 100 pistons, but now setting as initial weights, those we learned with 15 pistons. Again, we had to discard the ones from the mixer. In Figure 34, we show the best results we obtained, only presenting the testing plot as the training one is very similar:

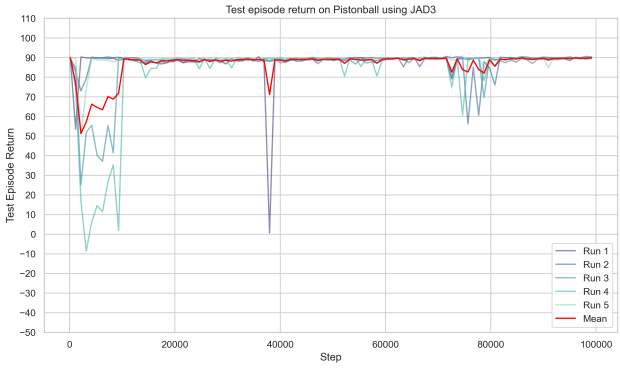


Figure 34: Pistonball continuous - 100 pistons - JAD3 - Fine-tuning

The goal was to see if we could fine-tune the policy we learned with 15 pistons and make the final push to a return of 92. Nonetheless, we could not, meaning the continuous version probably cannot surpass 90 because of the more fine-grained actions. Hence, fine-tuning the policy was useless, as it already achieved the best return it could.

We obtained those results by keeping a gradient clipping value of 0.5, reducing the learning rates to 0.0005, and the initial σs to 0.1. We expected to get better results with such small values because we did not want to change the initial policy a lot, which was already pretty good. We only needed to learn the mixer and adjust the weights of the other networks a little bit.

Still, we wanted to render the first testing episode to see if the fine-tuned policy would solve it differently than directly executing the initial one. Nonetheless, it solved it identically. When training from scratch, it learned a slightly different way to solve that episode. However, in all three cases, they solved the 25 testing episodes, on average, in 40 steps, achieving an average return of 90.

As with the discrete environment, we have not included any of the renderings of the testing episodes because, with 100 pistons, it was a little hard to see it with frames. Thus, we recommend looking at the GIFs we uploaded to our GitHub repository.

We have demonstrated again how we can use a policy learned in a task to solve a more complex one (i.e. increasing the number of agents) without further training, compared to a single-agent method like TD3, which already failed in the small task. It is true that even when fine-tuning, we could not improve the policy, considering how the ball stopped some times. However, it must be because the continuous actions lift/descend the pistons less than the discrete case.

7.3 ManyAgent Swimmer 2x2

This environment has two agents, each applying torque to two hinges. The action is a 2-dimensional vector taking values in the $[-1, 1]$ range. The observation is a 4-dimensional vector containing the agent's hinges information, and the state is a 12-dimensional vector containing all the hinges and sliders information. Moreover, the observation and state are unbounded. Unlike the previous environment, the episodes here are infinite, but we cut them after 1,000 steps.

We executed this 2x2 version because FACMAC did in their article. However, we found it strange that they could only achieve an episodic return of around 250, considering how the original single-agent Swimmer with one agent and two hinges (i.e. 1x2) could obtain 360. A longer swimmer should move further and get a higher return.

We investigated the problem by executing TD3 on the 2x2 Swimmer, and by increasing the discount factor γ, which we will later explain why, we could obtain a return of around 460. The main difference with FACMAC was that higher γ and the observations lacked certain information from the state. Nonetheless, that missing state information should come through the mixer network when computing the joint action-value function. However, even with the higher γ, the best policy any multi-agent method could achieve was the sub-optimal of around 250.

For this reason, we tried removing some of that missing information from the state and training TD3 again. We discovered that the slider's linear velocity was essential to achieving the optimal policy. The environment rewards how much the Swimmer moves in the right direction, using the non-controllable segment's tip as a reference position. As the linear velocity represents how much the position has changed, especially in the x-axis, we can see it as a hint to maximize the return.

Moreover, the people in Farama who currently maintain MAMuJoCo modified the environment's code to add the sliders' information in the observations by default, confirming our beliefs about needing that information [9]. We could have used that version, but as we wanted to compare our new method with FACMAC's reported results, we decided not to add that information.

In Figure 61, we present a sequence of the first episode frames to show how the Swimmer swims using the 250-return sub-optimal policy. The first agent's first segment does not move because the hinge has reached its maximum rotation but keeps applying torque. The non-controllable segment does not move either and points backwards, both segments slowing the Swimmer. Nonetheless, the other three controllable segments move coordinated.

We can compare it with the optimal behaviour TD3 achieves when using the complete state. We show it as another sequence of frames in Figure 60. The two segments of one agent are mostly opposite, and they need to coordinate with the other agent's segments. Besides, the two agents act identically, and in the ideal situation, we should not need agent identifiers even though we share weights when training the networks in a MARL method.

As explained in the previous environment, all the continuous methods use neural networks based on linear layers without leveraging any recurrent structure. We tried adding previous transitions to see if they could help. However, they did not. We do not know if this would help, as the MARL methods cannot learn the optimal policy, and TD3 leverages the fully observable state, not needing them either.

To reproduce and compare with FACMAC's results, we ran the methods for four million steps using one environment to gather data, as they did. However, running that many steps lasted more than the allowed time in a BSC POWER9 job. For this reason, we ran eight parallel environments. However, we were updating the networks the same amount as if we ran a single one, still exceeding the limit time.

As we have explained, we designed our training loop to run one or more episodes using parallel environments. Once they finish, we would update the networks as many times as steps executed in those episodes. We reduced the simulation time of interacting with the environment. However, we still spent lots of time updating the networks.

We solved the problem by decreasing the number of updates (i.e. mini-batches we sample) we perform after completing those parallel episodes, being a configurable parameter. We decided to perform 1,000 updates after the 8,000 steps (instead of 8,000), reducing the total updates to 500,000 instead of 4 million. With that, we could train any method for 4 million steps in the time limit of a BSC job.

[9] https://robotics.farama.org/envs/MaMuJoCo/ma_swimmer/

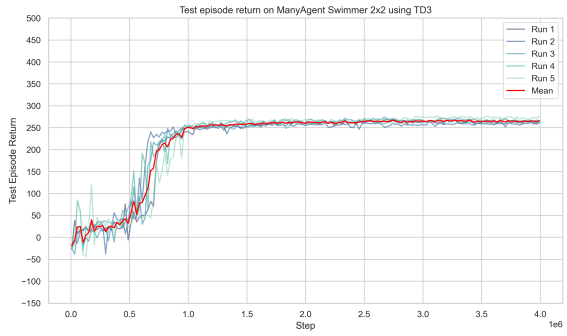
7.3.1 TD3

Even though the environment is initially multi-agent, we have one policy to generate the actions for all the controllable hinges using the environment state as input. As the states and actions are low dimensional, we expected this method to work better than the multi-agent ones, even if we remove the features that help reach the optimal performance from the state. Nonetheless, we would expect to work worse when increasing the Swimmer's length.

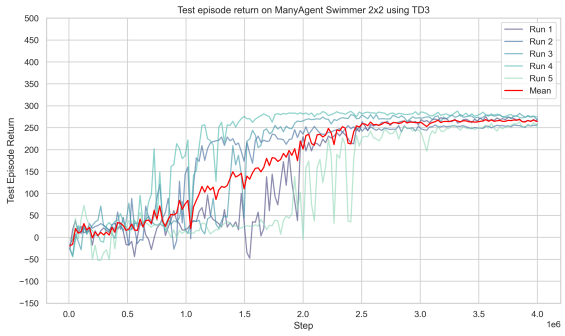
When running different hyperparameter configurations, we encountered a short article about the effect of the discount factor γ in the original Swimmer [11]. The author explained that using the usual 0.99 was not enough to reach the maximum performance, but it was necessary to bring it even closer to 1 (e.g. 0.9999). He explained that 0.99 was too small to consider the furthest steps, resulting in behaviours that only focused on moving properly during the first steps but usually got stuck during the end.

Hence, we ran our experiments with two values, the usual 0.99 and 0.9999. As we wanted to compare the results with the multi-agent methods, we ran TD3 with eight environments during 4 million steps, updating the network 1,000 times after completing those eight episodes. However, independently of the γ, we could only achieve the best sub-optimal policy the MARL methods can learn, which gets a return of around 250.

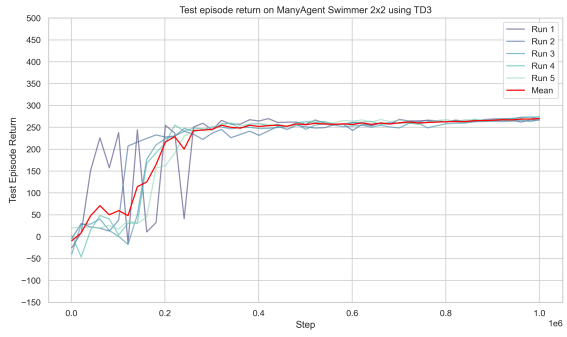
As running 1 million steps with one environment was affordable, we repeated the experiments with the two values of γ. TD3 did not learn the optimal policy when using the smaller one, but it could when using the bigger one, like [11] explained. In Figure 35, we show the best results we obtained with the different values of γ and parallel environments:



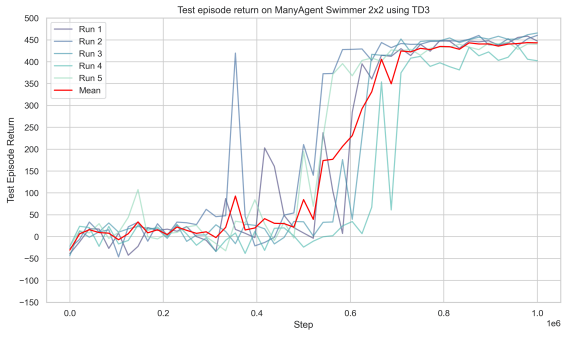
(a) 8 environments, γ=0.99



(b) 8 environments, γ=0.9999



(c) 1 environment, γ=0.99



(d) 1 environment, γ=0.9999

Figure 35: ManyAgent Swimmer 2x2 - TD3 - Changing γ and the number of parallel environments

We believe the problem was that we did not update the networks enough times to focus on the relevant state features, and we did not gather sufficient data considering how we ran eight episodes with the same policy, filling the replay buffer with similar data. We tried different tricks to keep the eight environments, like increasing the buffer size considerably to store more data. However, the results across seeds were unstable, not always reaching optimal performance.

One run in the results obtained when training with eight environments and the bigger γ gets a return of around 280, way higher than the 250 we have been saying. Nonetheless, we keep including it as the same sub-optimal policy with the flaw of the first agent's first segment not moving. Hence, even if the segments that move do it a little better, it still suffers the same problem, and we consider it the same. For this reason, we will see in different methods how even if the maximum return does not surpass 240, we will consider it the same policy.

Other hyperparameters we find worth mentioning are those related to the noise we add to the exploratory and target actions. We found that using an initial σ of 0.5 in the exploratory and 0.2 in the target ones was the best option. Nonetheless, the four configurations we show above used different minimum values, meaning they needed more or less exploration at the final stages of training. Besides, clipping the noise we added to the target actions to 0.5 was again the best option in all four cases.

Training with one environment for 1 million steps updates the networks 1 million times, compared to the 500,000 times we update them when using the eight environments. In that case, comparing the results trained with one with those trained with eight is a little unfair. Hence, we will use both when comparing with the MARL methods.

7.3.2 JAD3

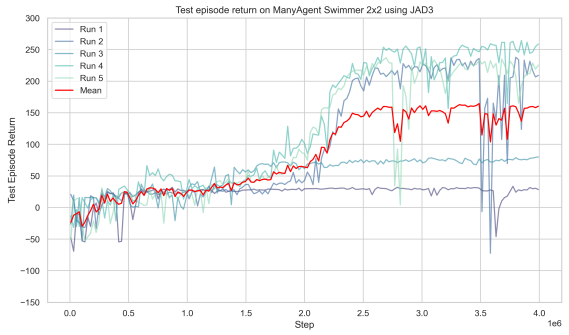
As we did in the Pistonball environment, we will compare the effect of updating the actors using the individual or joint action-value function, now depending on how many networks we train and if we use agent identifiers. Before that, we will explain some important hyperparameters to reach our results, the majority sharing value in all combinations we will compare.

First, setting an initial σ of 0.5 to generate the noise we add to the exploratory actions was the best. Nonetheless, we found a smaller initial value of 0.2 worked better for the target ones. Besides, clipping the target noise when the magnitude is over 0.5 gave the best results.

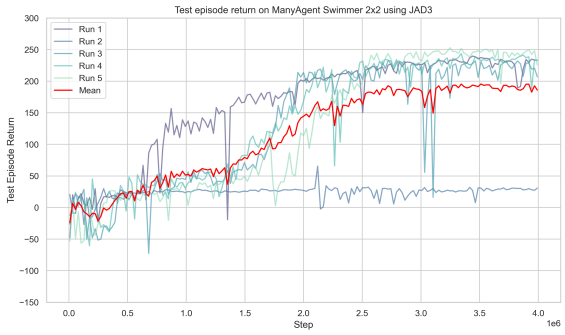
Another interesting parameter is the frequency at which we update the actors and the target networks compared to the critics and the mixers. We obtained better results if we updated all at the same rate. We believe the difference is because we used eight environments, reducing the number of updates we make. Hence, with those few updates, updating even less the actors and the target networks gave worse results.

We used the default values for the rest of the hyperparameters, except the learning rate in some cases, where a smaller value of 0.0005 worked better than the usual 0.001 to update the actor or the critic plus mixer. Moreover, even if we trained with eight environments, we achieved our best results using a buffer size of 500,000 transitions without suffering from having to fill it with similar data.

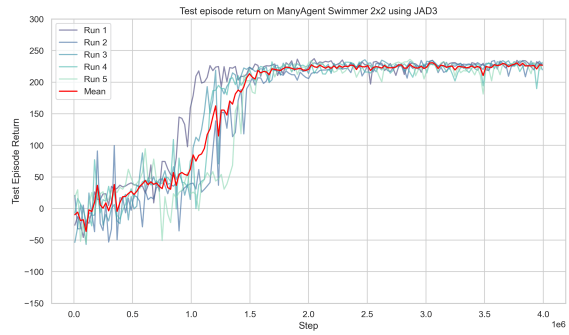
We can move to compare the effect of updating the actors with the individual or joint action-value functions, starting with training a shared actor and critic and showing the different combinations in Figure 36:



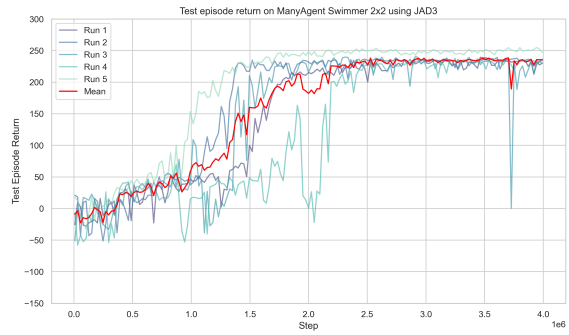
(a) Without identifiers
Updating the actor with $[Q_a]_{a=1}^N$



(b) Without identifiers
Updating the actor with Q_{tot}



(c) With identifiers
Updating the actor with $[Q_a]_{a=1}^N$



(d) With identifiers
Updating the actor with Q_{tot}

Figure 36: ManyAgent Swimmer 2x2 - JAD3 - Shared actor and critic

We obtain worse results when not adding the identifiers, the worst being when we update the actor with the shared individual value function. One or two runs do not learn, and those that learn do it slower and are less stable. Without identifiers and training a shared actor and critic, we cannot distinguish the inputs of each agent. Thus, we should not be able to learn different policies or individual value functions for each agent.

However, it does not seem to be the case in those results, as we still achieve a maximum return of around 250, meaning the first agent does not move its first segment. If we learned the same policy for both agents, they should behave optimally like the second agent or sub-optimal like the first.

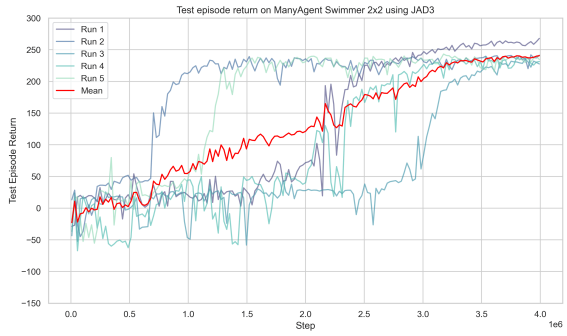
That segment does not move because its hinge cannot rotate anymore but still applies torque in the same direction. Hence, the observation for that hinge will consist of a maximum angle and a zero derivative, which the other hinges never face. With that, the actor and critic can distinguish the agents without identifiers by only looking at the observation, outputting different actions and value functions. Nonetheless, the network sometimes fails to focus on that, as some runs do not learn.

Without identifiers, two runs do not learn when updating the actor with the individual action values, compared to the joint one, which only has one. One explanation is the critic does not learn to distinguish the agents, and the method does not learn a proper decomposition of the joint function that gives different action values to each agent as they contribute differently to the joint reward. In that case, updating the actor with the individual values confuses the policy updates because the critic has not learned anything useful compared to the mixer.

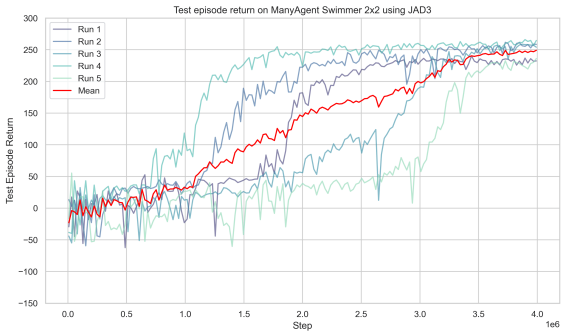
Adding the identifiers works well in both cases, validating the idea of learning different behaviours for each agent. Besides, updating the actor with the individual action values works better now, meaning the critic learns different action-value functions for each agent, helping the actor learn faster than using a single scalar, which hinders the updates. All runs learn fast and stable. On the other hand, using the mixer's output causes the different runs to learn at different and slower paces, even with some serious decays.

With the identifiers, the actor has an easier time learning different policies for each agent. The same goes for the mixer and the critic, as the method can learn a proper decomposition into different individual functions that represent the contribution of each agent to the joint reward. Hence, using the joint values confuses the policy updates because it cannot distinguish which agent's action was responsible for that high return, slowing learning.

Let's move to train independent actors but share the critic. We will follow the same procedure, reminding that the identifiers now apply only to the critic but not the actor, as it is not shared. In Figure 37, we show the results:



(a) Without identifiers
Updating the actor with $[Q_a]_{a=1}^N$



(b) Without identifiers
Updating the actor with Q_{tot}

(c) With identifiers
Updating the actor with $[Q_a]_{a=1}^N$

(d) With identifiers
Updating the actor with Q_{tot}

Figure 37: ManyAgent Swimmer 2x2 - JAD3 - Independent actors

Compared to sharing both actor and critic, even without identifiers, all five runs now learn the best behaviour, even though some need more time to learn. Besides, updating the actor with the joint values again gives better results, as most runs learn faster and reach a higher return. However, the difference is not big if we compare the mean return lines.

We are now training two actors and one shared critic, and without identifiers, the critic cannot distinguish the agent inputs. The only way would be to leverage the different observations because of that stuck first segment. Therefore, we might have the same problem of the critic not distinguishing the agents, not learning a proper decomposition of the joint value function into different individual functions, and explaining why using the mixer's output works better.

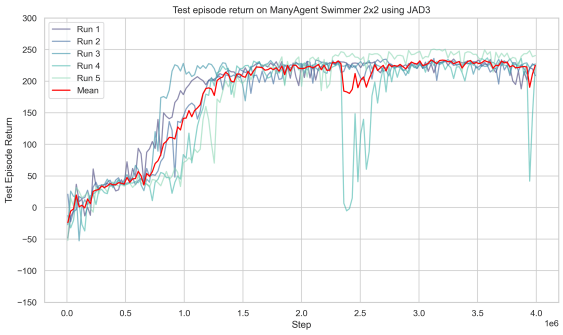
However, the results are better than when sharing weights, even when using the individual values to update the actors. We believe it may be thanks to having different actors learning separate policies, which can facilitate learning a decomposition of the joint function into different individual functions the shared critic outputs for each agent. Nonetheless, using the mixer's output is still slightly better because the critic is shared.

If we look at the results using the identifiers, they work better, and again, the mixer's output seems better. There is one run that takes more time to learn, and one decays just at the end. However, one run is stuck at 200 episode return when updating the actor with the individual functions. We can also look at the mean line to see how it learns faster when using the joint function.

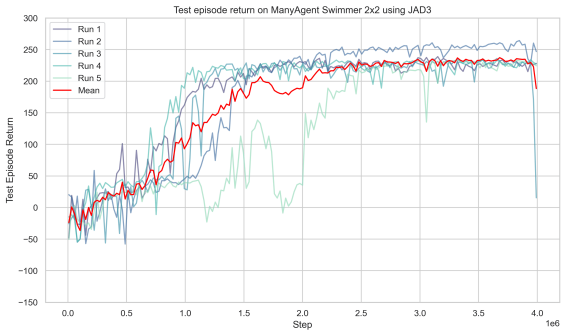
We simulated the environment to see the policy that got stuck in 200 episode return, showing the first frames in Figure 62, and the first agent was not moving any of the two hinges, both stuck at maximum rotation. Probably, the learned decomposition does not care about the first agent's doing, and all the reward contribution comes from the second agent. The episode return still increases, so the first agent's policy does not need to change, which is already stuck on maximal-torque actions.

We can conclude that the joint value function works better in this second variation, even though the difference is not that big. Maybe the problem here is only sharing the critic, resulting in a bad factorization of the joint function that hinders policy learning. For example, what we have seen about the first agent not doing anything is caused by learning a one-sided contribution. And even though we have identifiers to distinguish the critics' output, they are not always the invincible option that always works.

The third variation is training independent critics but sharing the same actor, adding the identifiers only to the actor. We show the results in Figure 38:

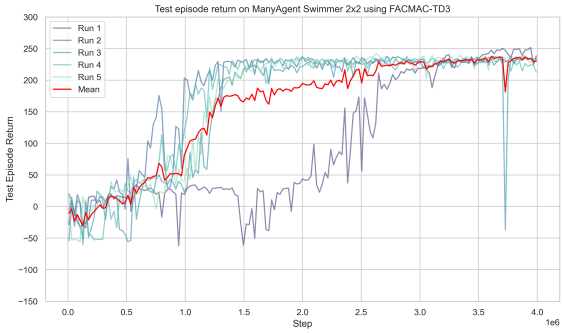


(a) Without identifiers
Updating the actor with $[Q_a]_{a=1}^{N}$



(b) Without identifiers
Updating the actor with Q_{tot}

(c) With identifiers
Updating the actor with $[Q_a]_{a=1}^{N}$



(d) With identifiers
Updating the actor with Q_{tot}

Figure 38: ManyAgent Swimmer 2x2 - JAD3 - Independent critics

Having independent critics seems the best option regarding performance stability across the four combinations. Almost all runs in all four combinations learn the best possible policy. Only one in the second plot seems to decay at the very end. Besides, there is now more difference in how we update the actor rather than if we use or not the identifiers.

When we use the mixer's output to update the actor, we have a run in both cases that learns slower than the other four. However, when using the individual value functions, all learn pretty fast, reaching 200 episode return in less than 2 million steps, the fastest only needing 1 million. Some runs in both cases indeed have some decay, but all recover very fast.

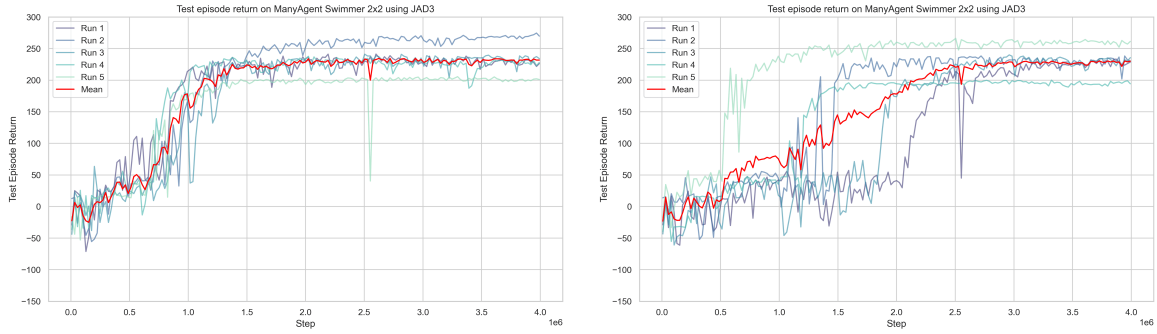
We believe there is less difference in using identifiers due to training independent critics and learning a proper decomposition of the joint reward that results in two different individual value functions. We have a shared agent, but because of the first agent's particular observations due to that hinge not rotating, the actor can still differentiate both agents.

We have more obvious differences when updating the actor with the joint or individual functions because the actor still needs to learn to differentiate from a single scalar each agent contribution and update the shared but different policy for both agents accordingly. Nonetheless, the difference is still low because we only have two agents.

Hence, we obtain better results using the individual action values to update the actor. If we had to decide if we use identifiers, we would have to consider if we want to reuse the policy with more agents.

With identifiers, the runs learn slower but have fewer oscillations near the end. On the other hand, without them results in faster learning, but the runs oscillate. However, without identifiers, we can reuse that policy and increase the number of agents.

Let's end with training independent actors and critics, where the agent identifiers do not have any use. Therefore, we only show the results when updating the actor with the individual or joint action values in Figure 39:



(a) Updating the actors with $[Q_a]_{a=1}^{N}$

(b) Updating the actors with Q_{tot}

Figure 39: ManyAgent Swimmer 2x2 - JAD3 - Independent actors and critics

Updating the actors with the individual value functions works better. Four runs learn in less than 1.5 million steps, compared to using the mixer's output, where four learn at different paces. In both cases, we have a run stuck on 200, meaning the first agent is not doing anything and does not contribute to the reward.

Having learned separate actors and critics, using the joint action values to update the actors seems, and is, inefficient. With the two critics, we can again learn a proper decomposition of the joint return into different individual functions, representing the contribution of the two agents. If we use them to update the corresponding policy, each agent will learn faster to act as expected.

However, if we use the joint action values, both actors will have to disentangle from one scalar if their action was good or maybe was useless but still achieved a high return thanks to the other's action. The second agent will probably perform fewer wrong updates as it contributes more if we consider how when the first agent does not move any hinge, the return is still 200. However, the first agent may take more time to learn its policy as it will see less variated data.

Regarding those runs reaching an episode return of 200, the problem is training separate actors. We believe the first agent rapidly gets stuck in producing maximum torque for both hinges. Then, the mixer and the critics will have learnt a decomposition that focuses on the second agent, the first agent's critic learning to produce high action value only for those maximal-torque actions, not allowing to modify the policy anymore to move the second hinge.

After analyzing all four variants, we can confirm that updating the actors with the individual value functions works better. It learns faster and is more stable. We only preferred using the mixer's output when training separate actors, but a shared critic. However, the difference was not that big.

Finally, we should pick the best results to compare with the other methods. Looking at the plots, training independent actors and critics, and updating the actors with the individual action-value functions seems the best option regarding performance and stability across the runs. Moreover, we can use

it to demonstrate how we can increase the Swimmer's length (i.e. the number of agents) and use the same policy without further training, as we do not have agent identifiers.

The two agents do not behave identically as they ideally should if we could learn the optimal policy. However, we believed that if we added more agents, all except the first would behave identically. For this reason, we created an architecture consisting of two sub-networks: one to handle the first agent and one shared to control the rest.

All the configurations we have shown until now, where we trained separated networks, were trained using that architecture. With two agents, there is no difference. However, thanks to that architecture, we could increase the number of agents and reuse the learned policies to demonstrate the MARL methods' scalability. In the next environment, which is the same but increases the number of agents to ten, we also trained this method leveraging that architecture, confirming our beliefs.

7.3.3 FACMAC

As we have explained, FACMAC's authors tested this environment to demonstrate how it could improve MADDPG. Nonetheless, they obtained sub-optimal results because the observations lacked important information to learn the optimal policy. For this reason, even though they trained the method for 4 million steps using one environment, updating the networks more times than we did with TD3, they could not learn a better policy that could surpass the return of 250.

As we could not train for that many steps with a single environment, we used eight instead and reduced the number of updates after the eight episodes from 8,000 to 1,000. We performed our hyper-parameter study, and in Figure 40, we show the best results we could obtain:

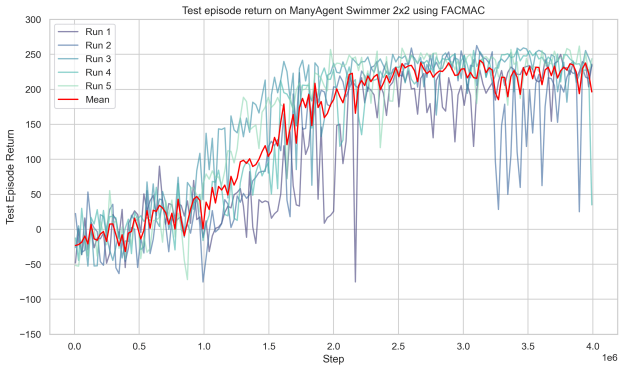


Figure 40: ManyAgent Swimmer 2x2 - FACMAC

We obtained these results by training a shared actor and critic without identifiers. Compared to JAD3's results from Figure 36b, FACMAC's results are better because all the runs learn, even though all suffer from high oscillations, especially the run breaking down at the end. FACMAC does not have the TD3 improvements, which should suffer more from overestimation. Overestimation happens by looking at how the runs oscillate, but for some reason, they help discover better actions faster.

Even though we used eight parallel environments, we did not increase the buffer size proportionally but used the same size as the previous method. Another interesting parameter is the Gaussian's σ, in which we set an initial value of 0.5, like our new method, and decreased it until 0.05 during training. We used default values for the learning rates, network sizes, and gradient clipping.

If we compare our results with those reported in FACMAC's article, we achieved similar results even though we updated the networks fewer times. Their mean return across the different seeds fluctuated around the 200 episode return, having a performance breakdown near the end the method could more or less restore.

On the other hand, our results seem better and more stable, especially if we look at the mean episode return. Once the method learns, it remains decently stable over the 230 mark, even though some runs oscillate during all training. The problem is that one run crumbles just at the end of the training, lowering the mean return to under 200, not knowing if it would recover.

We simulated how the best policy FACMAC learned solved the task, and it learned to move all the hinges except the first one of the first agent. It achieved a return of around 250, so we consider it to have grasped the same sub-optimal policy JAD3 and TD3 learned. Thus, as we have previously shown how it behaves, we do not add the frames again.

7.3.4 IQL

We also trained this environment with the continuous version of IQL to see if training fully decentralized could solve the task. We already saw how it suffered in the previous one, and we would expect IQL to fail due to the increased complexity and coordination needed. The hinges must coordinate their rotation to move the segments optimally, especially considering how the segments are connected, unlike the pistons that acted individually, and how a hinge rotation affects the whole Swimmer.

We started our experimentation with the usual hyperparameter study, training with eight parallel environments during 4 million steps. However, IQL could not learn any decent policy independent of the configuration we used, with the mean line oscillating around 0 most of the time. Some runs had sudden peaks of 100 but always returned to 0. The best one we could achieve was one that seemed to start improving near the end, meaning the networks did not update enough times.

We simulated the testing episodes with that best run to see what IQL had learned. We obtained those results by training separate actors but a shared critic (without agent identifiers) and, as in previous methods, it presented the same problem of the first agent's first segment not moving due to its hinge being at maximum rotation, and the non-controllable segment pointing backwards. The other segments were moving a little, as if they had spasms, without coordination and the Swimmer not advancing almost anything.

We were unsure if the problem was due to the inherent limitations of IQL of training fully decentralized, not learning any coordinated behaviour, or if we were not updating the network enough times. For this reason, we repeated the hyperparameter study, now training for 1 million steps with a single environment. The results were a little better, but still not comparable to the ones from previous methods. In Figure 41, we show the best results we obtained:

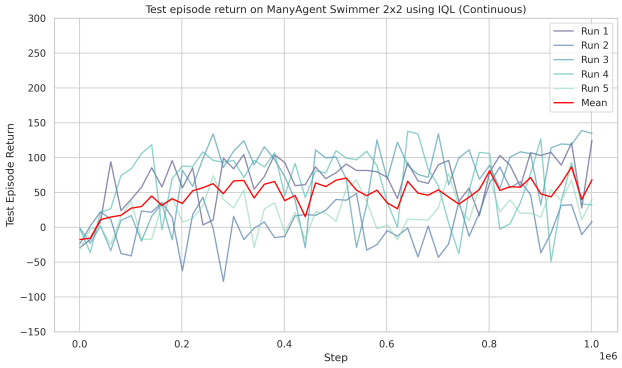


Figure 41: ManyAgent Swimmer 2x2 - IQL

We obtained these results by training independent actors and critics. It is true that when training eight environments, we obtained better results when sharing the critic, but the difference was small compared to when training with two. Moreover, using an initial σ of 0.2 worked better than the usual 0.5 we found with the previous methods.

We simulated again the environment with the best weights to see what IQL learned. Essentially, it learned the sub-optimal policy of 200 return but worse. The first agent was not moving any segment because it was applying maximum torque in the same direction, the hinges always being at its maximum rotation.

The second agent applied torques of different magnitudes and signs to the two hinges to move their segments. However, the torque was very high in some cases, moving the segments aggressively and inefficiently. For example, the non-controllable segment pointing backwards sometimes collided with the second agent's first segment due to applying maximum torque. Due to applying such high torques, the reward penalization term was higher.

Training the networks for more steps achieved better results, as the mean increased from being 0 all the time to now oscillating around 50. Nonetheless, we still achieved worse results than our new method or FACMAC, meaning the problem is due to IQL training being decentralized and ignoring the other agents.

It is true that with our new method, we ended up with that sub-optimal policy where the first agent was not moving any hinge. However, it was in a run out of five, with a particular configuration. We tried the same hyperparameter configurations with IQL but could not learn anything better. Hence, if the agents would not ignore the others while training, IQL could learn at least the best policy any MARL method can achieve.

7.3.5 MADDPG

FACMAC's authors already executed MADDPG in this environment to demonstrate how FACMAC could improve MADDPG thanks to the joint reward decomposition and the new actor update equations. Nonetheless, we also trained it because it is the only algorithm we found not decomposing the joint reward and working with continuous actions, so we used it to represent that method type.

As with IQL, we started our hyperparameter study by training the method for 4 million steps and using eight parallel environments, repeating the same problem of not training the networks enough in that many steps. Nonetheless, the best configuration's mean results were higher than 0 and closer to

50, meaning MADDPG could work better than IQL.

We decided not to simulate the results and directly repeat the experiments, training for 1 million steps using one environment, obtaining better results again. In Figure 42, we show the best results we obtained after studying the hyperparameters:

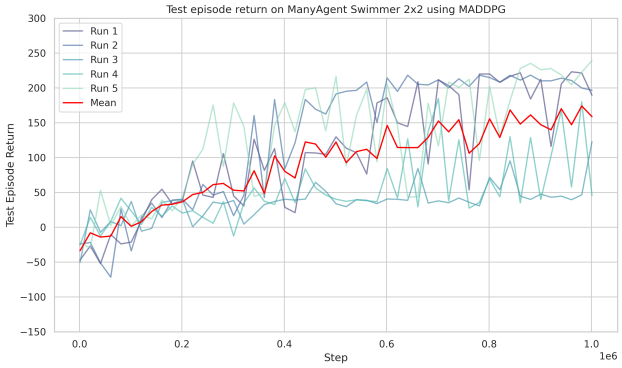


Figure 42: ManyAgent Swimmer 2x2 - MADDPG

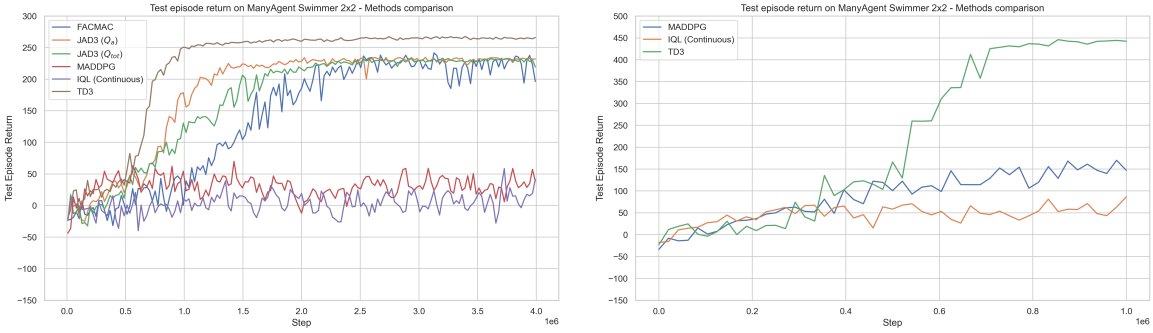
We obtained these results by training a shared actor without adding agent identifiers. The best initial σ for the Gaussian noise was 0.2, with a gradient clipping value of 0.5 in both networks and a slightly lower learning rate (0.0005 instead of the usual 0.001). Finally, we obtained better results by setting a bigger network size in the monolithic critic (using 256 units per layer) than the actor, which used the default 64 units.

We compared our results with those reported by FACMAC's authors. Considering their results until one million steps, we achieve similar mean results because some of our runs surpass a return of 200 while theirs only have sudden peaks. However, our results show more variation across the runs, with one decaying at the end below 100. Nonetheless, their results did not improve after the 1 million steps, worsening at the end and becoming worse than ours.

We simulated the environment with the best run, the one almost achieving a return of 250 at the end, to see if it also learned the same sub-optimal policy as the first two MARL methods, which it did. Moreover, we also simulated the testing episodes with the runs achieving a return of 200, and both learned the other sub-optimal policy where the first agent does not move any hinge, but the second one moves both optimally. Hence, we do not add the frames as we already presented them.

7.3.6 Comparison

Let's compare the best results achieved with the different methods, plotting the two update mechanisms with JAD3. As we executed some methods with a different number of parallel environments and duration, we will show two plots, presenting a method twice if we have run it with both settings. With IQL and MADDPG, we plotted the best results we obtained when training eight environments we did not present in their respective subsections. In Figure 43, we show the plots:



(a) 8 environments, 4 million steps

(b) 1 environments, 1 million steps

Figure 43: ManyAgent Swimmer 2x2 - Methods comparison

We have selected the best results of each method, independently of the scalability properties. With JAD3, we have plotted the best results of each update mechanism. Hence, when updating the actor with $[Q_a]_{a=1}^N$, we selected those we obtained when training independent actors and critics. When updating the actor with Q_{tot}, we selected those we obtained when training a shared critic but separate actors, adding identifiers in the critic.

TD3 achieves better results than all the MARL methods in both settings, which we expected considering how the actions are 4-dimensional and generating them from a single network should be easy. The state has information the observations do not directly see. However, even if we added that information, TD3 would still learn faster because of the extra difficulty of training a MARL method.

We did not add that information because we wanted to compare our new method with FACMAC's reported results, which they did not add. Thus, it would be interesting to repeat the experiments after adding that information and see if the MARL methods could learn the optimal policy. Besides, we would like to see if they would benefit or not from using the higher γ like TD3 did to learn the optimal policy.

If we compare our method with FACAMC, we can see how it learns faster and more stable, especially as training proceeds and near the end, while FACAMC always oscillates and decays at the end. Thanks to the TD3 improvements, our method suffers less from overestimating the action-value functions, which marks the difference between the performance both methods show, especially if we consider how JAD3 (Q_{tot}) is FACMAC with TD3's improvements.

If we compare if it is better to update the actor with Q_{tot} or $[Q_a]_{a=1}^N$ in our new method, we can see how it is better to use the individual action values. It learns faster the best policy, and the line seems more stable at the first stages with lower oscillations. After both learn the best policy, there is no significant difference, except the one updating with Q_{tot} decaying at the end.

IQL achieves the worst results independently of how many network updates we make. The problem with IQL is the decentralized training, which ignores the other agents and does not learn any coordinated behaviour, which is needed in this environment, even in the sub-optimal policy all MARL methods can learn.

It could work in the Pistonball environment, especially in the discrete version, because the pistons needed less coordination, and the actions of one affected the others less. Even with that, IQL solved the task sub-optimally when the actions were continuous because the pistons needed more coordination,

which IQL could not learn.

Finally, MADDPG works worse than the other MARL methods following the CTDE paradigm but at least better than IQL. Again, it worked better when updating the networks more times, which allowed us to learn the two sub-optimal policies with a return of around 250 and 200, the same as FACMAC and our new method. Nonetheless, the results of the other three runs were pretty bad, explaining the low mean line.

The problem of MADDPG is the centralized and monolithic critic, which hinders learning because it needs to learn to approximate the joint function from all the agent's actions and the hidden state. Even if FACMAC does not leverage the decomposition to update the actor, learning Q_{tot} from the critics' outputs is faster. Besides, MADDPG's actor update equation samples outdated actions from the replay buffer, hindering learning even more.

We can conclude that leveraging MARL methods to solve this environment was not worth it because the action dimensionality is very low, and the single-agent ones work better. Nonetheless, among the multi-agent methods, those following the CTDE and decomposing the joint action-value function worked better, and our new one could improve FACMAC.

7.4 ManyAgent Swimmer 10x2

This environment is the same as the previous one but increases the number of agents to ten. Thus, we have a longer swimmer, with 21 segments (20 controllable plus the first one), that moves further and reaches higher returns. The observation space, action space, and reward function are the same, and the state is now 44-dimensional because we have more hinges.

The observation still misses information, but as FACMAC also ran this version to report their results and we wanted to compare our new method with this longer Swimmer, we did not add it here either. Therefore, we faced again the problem of the first agent's first segment not moving and the non-controllable segment pointing backwards.

Nonetheless, in this 10x2 Swimmer, we expected the multi-agent methods to work better than TD3, even if we missed information, because of the increased action dimensionality. The problem, however, as we will later see, is that TD3 could not learn anything, so we do not know the maximum return an episode could reach.

Regarding the new method, we also performed the same study to determine if using the individual action-value functions to update the actor was better than using the joint one. However, we did not test all four combinations of training shared or independent actors and critics, but only those that did better overall in the 2x2 version.

Finally, we decided not to execute IQL and MADDPG because they already achieved bad results with fewer agents. Hence, we also expected bad results here. Moreover, we checked the results reported by FACMAC's authors with MADDPG, and they achieved even worse results than with the 2x2 configuration.

7.4.1 JAD3

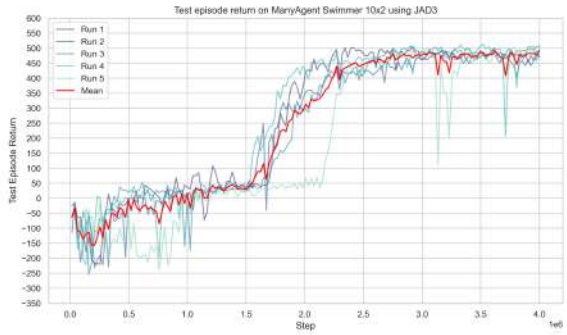
We studied the effect of using the individual or joint action-value function to update the actor, but only the variants sharing the actor but not the critic, and the one not sharing any network. As a reminder, when not sharing weights, we train two networks instead of one per agent. One controls the first agent, while the other controls the rest.

Regarding the hyperparameters we mentioned in the 2x2 case, we found the same values to work the best in this case, so we also used them. The only one worth mentioning, which we do not consider as a hyperparameter, controls if we compute the minimum of the two approximations when computing the actor loss, like SAC, or only use the first, like TD3. In the 2x2 Swimmer, we always computed the minimum.

When removing the minimum, no matter how many hyperparameters we tried, we could not match or improve our FACMAC's results. In most cases, the new method started to learn at the end, needing more than four million steps. We believe the algorithm was not exploring the action space enough, which makes sense considering how we now have to train two sets of critics and two mixers. We changed how the algorithm explores, reducing the speed at which the Gaussian's σ decreases (initially linear), but the results were the same.

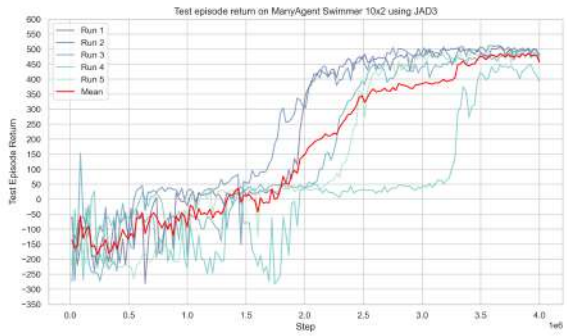
However, when computing the minimum again, with the initial linear exploration mechanism, we could achieve better results than FACMAC, learning the best policy faster, without those oscillations FACMAC presents. We removed the minimum because we wanted to explore if it had some benefit, as we added it on a whim when programming the new method. Fortunately, it was the key element to reach better results.

Let's compare the effect of updating the actor with the individual or joint action-value functions, starting with sharing the actor but training separate critics. In Figure 44, we present the results, also exploring the effect of adding the agent identifiers in the actor:



(a) Without identifiers
Updating the actor with $[Q_a]_{a=1}^N$

(b) Without identifiers
Updating the actor with Q_{tot}



(c) With identifiers
Updating the actor with $[Q_a]_{a=1}^N$

(d) With identifiers
Updating the actor with Q_{tot}

Figure 44: ManyAgent Swimmer 10x2 - JAD3 - One actor and two critics (first and other agents)

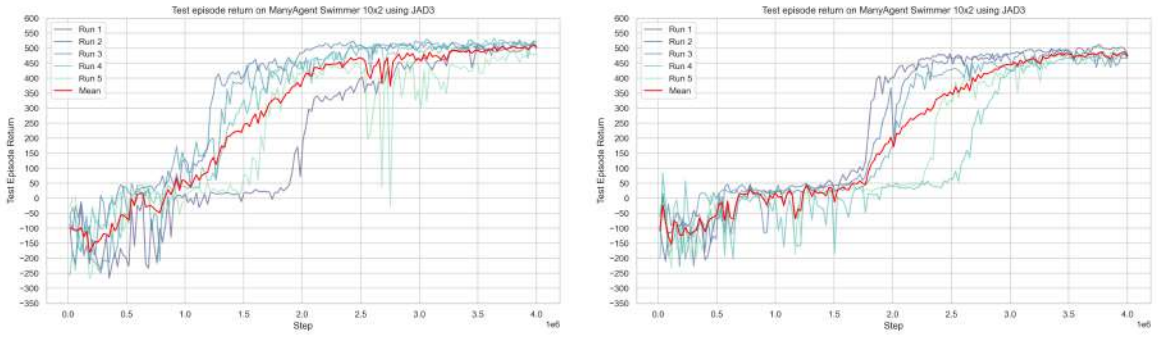
First, we can observe how all runs learn the best, or almost the best, possible policy, except for two having a sudden breakdown at the end. Second, the agent identifiers make a significant difference compared to the 2x2 case, and the four plots are more different. Third, using the individual action-value functions to update the actors works better, even though some runs learn faster when using the joint function.

Compared to the 2x2 version, with ten agents, using the identifiers slows learning. Maybe the identifiers are taking too much importance for the actor, considering how out of 14 features of the observation, 10 are identifiers. We believe those identifiers force the actor to learn a somehow different behaviour for each agent, even though most agents should behave identically.

More runs learn slower when updating the actor with the mixer's output than the 2x2 case. We have more agents, even though most act identically. However, determining the contribution of each agent to the joint reward from a scalar value is more difficult with more agents, confusing the policy updates and taking more time to learn the correct policy.

And the worst comes when combining the identifiers and the updates using the joint value function. We have the highest number of slow runs, and they are even slower due to the identifiers. The identifiers force the actor to learn different policies for each agent, and the policy needs to extract the contribution of each agent from a scalar action value. Nonetheless, the policy can still learn the best behaviour before the training ends.

Let's move to train two actors and two critics, following the same idea of one controlling the first agent and the other the rest. In Figure 45, we present the results using the individual or joint action-value functions to update the actor:



(a) Updating the actors with $[Q_a]_{a=1}^{N}$

(b) Updating the actors with Q_{tot}

Figure 45: ManyAgent Swimmer 10x2 - JAD3 - Two actors and two critics (first and other agents)

Updating the actor with the individual value functions works better, even though one run suffers a decay and takes time to recover. The mean line is steeper and reaches a return of 400 in 2 million steps. Besides that run suffering decay, there is one run that learns slower but still learns at a good pace. On the other hand, when using the mixer's output, all runs learn slower, as the plot is similar but shifted to the right. No run suffers decay, but the mean line is still worse, less steep, and never reaches 500.

We trained separate actors and critics, each learning a different policy and action-value function. In that case, updating the actor with the joint action values is inefficient, as each actor needs to extract each agent contribution from a scalar, causing wrong updates when the high value is due to the agent the other actor controls. It still learns because we only have two networks, but slowly. We believe if we had trained one network per agent, it would have given even worse results.

We simulated the testing episodes to see how the new method learned to solve the task, using the best weights of the different network architectures, but all learned the same policy, the best a MARL method could learn. All the controllable hinges were moving except the first agent's first one, like the one in Figure 61, but with a longer Swimmer. We tried adding the frames, but it was impossible to see anything because of the Swimmer's length. Thus, in all cases, we recommend looking at the GIFs we have uploaded to our GitHub repository.

7.4.2 FACMAC

Compared to the 2x2 version, achieving good results with FACMAC was pretty easy, even though the algorithm implementation was the same. We even surpassed those reported in the original paper, reaching an episodic return of 500, the highest a multi-agent method can achieve. In Figure 46, we present the best results we obtained:

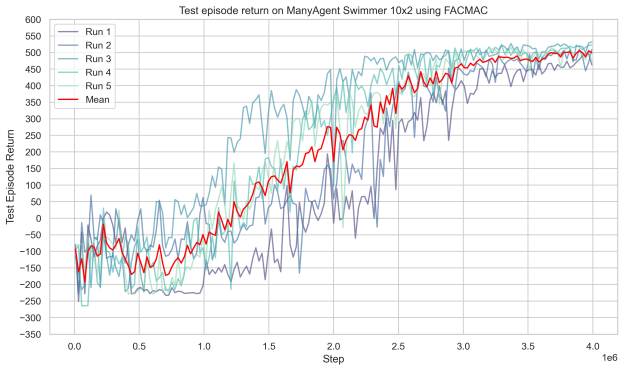


Figure 46: ManyAgent Swimmer 10x2 - FACMAC

We obtained these results by training two actors (i.e. one to control the first agent and the other the rest) and a shared critic without agent identifiers. In the 2x2 version, we obtained the best results by sharing both networks, not using identifiers either.

With this one, we also obtained very high results by sharing both networks, but they were slightly better when training a different actor for the first agent. As the first agent behaves slightly differently, separating the actor helped avoid updates that could deteriorate the agents' policies.

The other hyperparameters are mostly the same as the best ones in the 2x2 version, especially the initial Gaussian's σ, the gradient clipping, and the network sizes. The only difference is the learning rate in the critic is slightly lower than the actor's (0.0005 vs 0.001).

We can see how FACMAC learns the best policy in all runs, and the results are more stable as training proceeds compared to the ones we obtained in the 2x2 version. The reason might be having more agents behaving identically, easing learning the policy of those moving both hinges and giving more time to the first agent, the one moving a single hinge, to explore better actions and avoid getting both hinges stuck, as we saw in the 2x2.

As we update the actors with Q_{tot}, even if we share a critic and do not use agent identifiers, FACMAC should not suffer that much if we cannot learn a good decomposition that assigns the correct contribution to each agent, as long as we can correctly approximate the joint action-value function. Hence, having more agents should also help learn the critic faster because even if the first agent behaves slightly differently, the actions that achieve higher joint rewards should be similar because the first agent

will have to move its second hinge like the other agents.

If we compare our results with the reported by FACMAC's authors, they achieved a mean episode return of around 430 in the 4 million steps they trained using a single environment. Hence, we achieved better results while updating the networks fewer times. Nonetheless, they also achieved better results across the different runs than the 2x2 version, even though their best hyperparameters were mostly identical, confirming that having more agents facilitates training the task.

We simulated the testing episodes to see how our FACMAC learned to solve the 10x2 Swimmer. As the previous method, it achieved the best policy any MARL method can learn, where the first agent still does not move its first segment, but all the others move.

7.4.3 TD3

As mentioned, TD3 could not learn anything, no matter how many combinations of hyperparameters we tried. In Figure 47, we show an example of the obtained results across the five seeds. We trained using the same strategy as in the 2x2 version, using one environment and training for one million steps. We also tried the different values of γ, but there was no change.

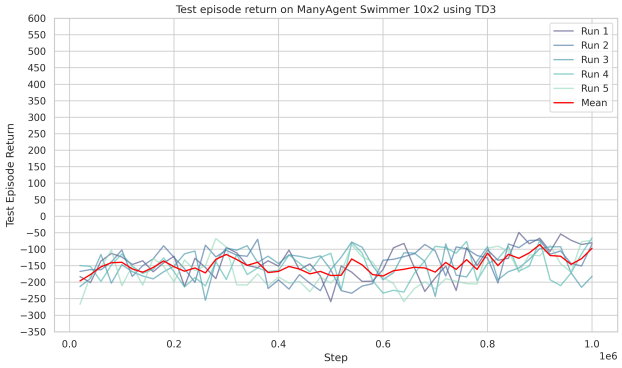


Figure 47: ManyAgent Swimmer 10x2 - TD3

The problem is the increased action dimensionality, going from 4 to 20, each taking values in the $[-1, 1]$ range. It is not that the networks need to see all the possible values in that range for each sub-action, but at least some values. We have a single actor outputting 20 sub-actions from the fully observable state, resulting in an exponential combination to explore to learn the optimal policy.

Therefore, in the same steps we learned in the 2x2 version, we could not grasp any good policy in the 10x2, as we expected. Maybe training for more time could result in something. However, considering how it also failed in the Pistonball environment, which was less complex and with fewer actions, it would probably not succeed.

Nonetheless, we rendered the testing episodes to see what was happening even if it did not learn anything. We observed almost all hinges blocked at their maximum rotation, still applying torque, and the Swimmer folding like an accordion until it stopped moving any hinge.

7.4.4 Comparison

Let's end by comparing the best results obtained in the different methods. We present the plot comparing them in Figure 48. However, we did not add TD3 because we trained it with one environment for 1 million steps, and considering how it did not learn anything, it was not worth putting it again as

a separate plot.

With JAD3, we plotted the best results obtained with each update mechanism. When updating the actor with $[Q_a]_{a=1}^N$, we selected those obtained when training separated actors and critics. On the other hand, when updating with Q_{tot}, we chose the results obtained when training a shared actor without identifiers and separate critics.

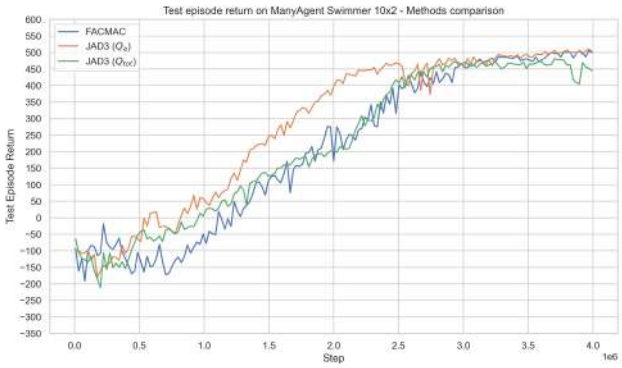


Figure 48: ManyAgent Swimmer 10x2 - Methods comparison

As in the 2x2 Swimmer, JAD3 learns faster than FACMAC, at least the version that updates the actor with $[Q_a]_{a=1}^N$ until both methods learn the optimal policy, where both are very similar, but FACMAC oscillates a little more. Nonetheless, JAD3 obtains similar results to FACMAC's when we update the actor with Q_{tot}. JAD3 learns faster at the beginning, but after some time, the results are very similar until the end when our new method suffers a decay in one run.

Our method has the TD3 improvements that help avoid the overestimation problem. They help because the two plots show fewer oscillations, especially the one updating the actor with the individual action-value functions, which makes fewer wrong updates compared to using Q_{tot}.

As FACMAC oscillates a lot, the curve is higher at some points than JAD3 (Q_{tot}). Considering how it still learns the best policy, the overestimation problems are helping it reach higher sudden peaks later counteract because it also suffers sudden decays. However, as training proceeds, the oscillations diminish.

We can conclude this experiment by saying that leveraging MARL methods was essential to solve the task because TD3 could not learn anything in that many steps because of the high action dimensionality. We did not run MADDPG and IQL because we already demonstrated in the smaller version that they worked worse. Moreover, JAD3 (Q_a) obtained better results than FACMAC.

7.4.5 Scalability

We wanted to test the scalability properties of the MARL methods in the ManyAgent Swimmer environment, repeating the same process we followed in the Pistonball. We decided to reuse the weights we learned with the 2x2 version and increase the number of agents to 10, leveraging the results we already obtained when experimenting with the 10x2 version. Thus, we would only need to fine-tune, but we already trained from scratch.

We decided again to use the MARL method with the best results, which was JAD3, but updating the actors with $[Q_a]_{a=1}^N$. We started by executing the 10x2 Swimmer using the weights learned in the 2x2 without further training. We were unsure about which set of learned weights to use, as we needed

one not using agent identifiers that could have learned the best policy.

We first used the ones where we trained two critics (one controlling the first agent and the other the rest) and one actor. The Swimmer started moving all but the first segment. However, because of training a shared actor and the first agent in the 2x2 learning not to move one segment, as the Swimmer kept moving, more segments started to stop because their hinges were at maximum rotation. It did not fully stop because some hinges recovered, and the segments moved again. However, in some cases, the Swimmer ended up rolled up.

Then, we loaded the weights where we trained two actors and critics, and the behaviour was pretty different. The first agent kept moving a single segment. However, the remaining nine moved decently, the hinges never blocking themselves. However, as we had a longer Swimmer, it started moving downwards instead of to the right, achieving an average episode return of around 170.

Hence, the next step was training the 10x2 version, leveraging the second set of weights we tried, where we trained one actor and critic to control the first agent and another to control the others. The idea was to see how we could fine-tune those weights for fewer steps than training from scratch and learn the best policy faster. Again, we had to discard the mixer weights.

We performed a hyperparameter study, trying a low initial Gaussian's σ to generate the noise and low learning rates. We achieved the best results when training the 10x2 Swimmer from scratch using a small gradient clipping value, so we maintained it. In Figure 49, we present our best results:

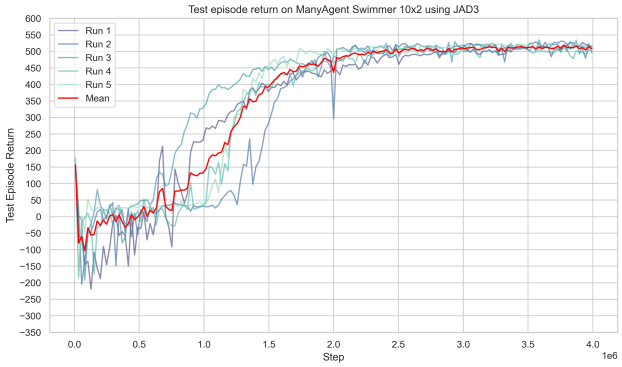


Figure 49: ManyAgent Swimmer 10x2 - JAD3 - Fine-tuning

We obtained these results by setting the initial Gaussian's σ for the exploratory actions to 0.2 and 0.5 for the target actions. Besides, we used a lower learning rate in the actor (i.e. 0.0005). Compared to the results when training from scratch, which were the ones we presented in Figure 45a, we have inverted the σs and lowered the actor learning rate.

If we compare the plots by looking at the mean line, we can see how leveraging the learned weights learns the best policy faster. Training from scratch got a return of 500 near the end. However, we now reach it in less than 2.5 million steps because the performance is more similar between the runs. On the other hand, when we trained from scratch, the runs learned at different paces, some learning the best policy near the end.

We demonstrated again how, thanks to the agents behaving identically, or a subset of them, we can learn first to solve a smaller version of the same task and fine-tune those weights to solve the bigger one faster, reaching the same policy. Nonetheless, compared with the Pistonball environment, the initial

performance was way worse, and improving it to the best policy took quite a time. However, we could expect it, given the higher complexity of this environment.

7.5 Ant

We wanted to use the Ant environment to demonstrate how we could leverage multi-agent methods to learn general policies that could solve similar tasks, compared to single-agent ones that should train on each independently. The Ant environment fulfilled our two requirements: allowing us to define similar problems and the robot having a symmetric structure the MARL methods could leverage. In Figure 50, we show a scheme to understand what we mean by similar goals and symmetric structure:

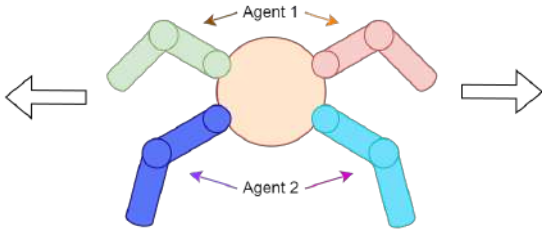


Figure 50: Ant - Problem formulation

The original task was to learn to control the legs for the ant-like robot to walk the furthest to the right as possible in the 1,000 steps an episode lasts. We could solve it using a single-agent method or divide it into multiple agents and leverage a MARL method. In that case, we could have, at most, four agents, each controlling a leg.

Let's assume we divide the environment into two agents, each controlling two legs as depicted in Figure 50. In that case, we could cut the robot horizontally in half, and both halves would be identical. If they are identical, they should behave identically. If you imagine the robot moving to the right, the red leg should move like the cyan and the green like the blue. Thus, a MARL method could train a single shared actor and critic to learn to control both pairs of legs.

That is what we meant by a symmetric structure a MARL method could leverage. On the other hand, a single-agent method that learns to control all the legs at once would not leverage that symmetric structure because it does not matter how we order the legs because a single agent is controlling them all at once.

The next requirement was defining similar tasks. The original goal was for the robot to walk to the right. However, we can control the direction by changing the term in the reward function that gives a positive value for how much the robot has advanced to the right since the last action. Hence, we could change that term to give a positive reward when advancing to the left.

Therefore, our objective was to demonstrate how we could learn a single shared policy to control the agents such that we could use it to make the robot walk to the left or right without rotating the robot. On the other hand, if we tried to learn a single policy to walk in both directions using a single-agent method, it would probably fail, making it easier to learn them separately.

Even though we could change the sign of the reward term to learn to walk to the left, it is not that simple to modify it to train to walk in both directions. We preferred keeping the original reward and applying a different trick. Hence, the robot would learn to walk in both directions by only exploring the action space while walking to the right.

Our solution was starting from Figure 50 and rotating the agents and the torso-related information that could give away the direction. Hence, the ant would walk to the right, but the information the networks would receive while training would be what it should expect when walking to the left (e.g. the torso's linear x-velocity increasing to the left).

Moreover, each agent would learn to control both pairs of legs independently of which leg is the forward and which is the backward. When walking to the right, we can consider the red and cyan legs as the forward legs, while the other two are the backwards. On the other hand, when walking to the left, the forward ones would be the green and blue legs. We present our solution in Figure 51:

Figure 51: Ant - Our approach

When moving to the right, the first agent controlled the red and green, and the second the blue and cyan. On the other hand, when moving to the left, the first agent controlled the cyan and blue, and the second the red and green. Without changing the reward function and only changing which legs control each agent, a MARL method should learn a policy to move in both directions without rotating the robot.

Nonetheless, we had to modify the MAMuJoCo wrapper that divides the initially single-agent environment into multiple agents. First, to learn to walk in both directions equally, we decided to alternate the episodes where the robot had to walk in each direction. We added a boolean flag to the observations and the fully observable state to indicate the current direction.

Second, we had to modify the observations such that each agent would receive the hinges' information it controlled at each episode. Moreover, we had to change the torso-related data when walking to the left during training. In particular, we reversed the linear and angular velocity (the xy coordinates) and rotated the quaternion.

Initially, the observation only contained information about the hinges the agent controlled. Nonetheless, similar to the previous environment, we had to add the torso linear velocity to learn the optimal policy, explaining we had to modify the torso-related information in the observations. Thus, we had to rotate the torso-related data in the observations and the fully observable state.

Third, we had to place the correct sub-actions in each position of the action vector. The environment is initially single-agent, the original action being an 8-dimensional vector. As each agent outputs the torque to apply to the four hinges it controls, we had to ensure we placed them according to the order of the single-agent action:

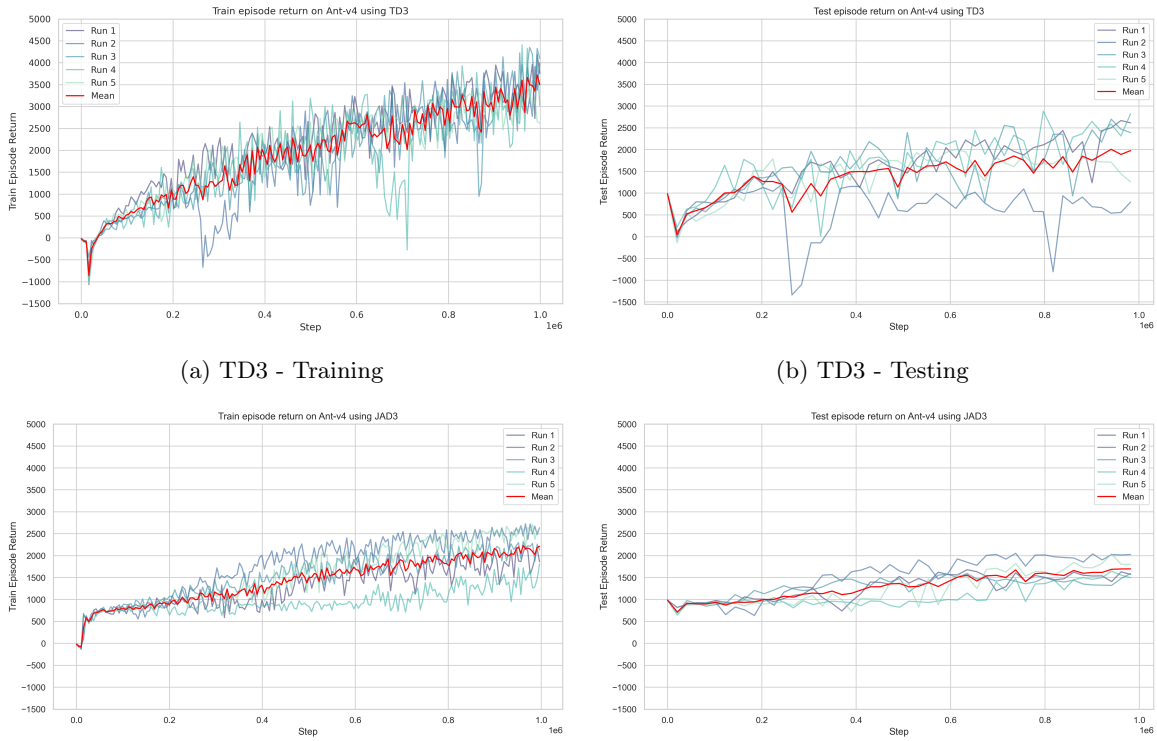
$$[hip_4, ankle_4, hip_1, ankle_1, hip_2, ankle_2, hip_3, ankle_3] \tag{55}$$

For some reason, they decided to put the fourth leg first. Hence, if the first agent controls the first and second leg, we must ensure we put the actions it outputs to the third, fourth, fifth, and sixth positions. The same happens with the second agent controlling the third and fourth legs.

With that, we would pretend to rotate the robot to learn to walk to the left. However, when simulating the environment and testing how the robot walks to the left, we would not apply the trick of rotating

the agents because the agents should have learned a general policy. Hence, we should not modify the torso information nor change which legs each agent controls, and only change the flag.

We only trained TD3 and JAD3 to demonstrate how MARL methods should learn more general policies independently of the method type. As we achieved the best results in most cases with our method, we used it as the representative. In Figure 52, we show the testing results we obtained with both:



(a) TD3 - Training

(b) TD3 - Testing

(c) JAD3 - Training

(d) JAD3 - Testing

Figure 52: Ant - TD3 and JAD3 results

The maximum return one could achieve in the original task was more than 5,000. However, a policy getting more than 3,500 is already very decent. In the end, what changes is the ant learning to walk faster, but we were satisfied if it could learn to at least walk in both directions. Besides, as it has to learn to walk in both directions, we expected to reach a lower return.

The training and testing results are quite different, especially in TD3. When training, we always run episodes where the robot walks to the right, meaning the reward is always positive. However, we expect the robot to walk to the left during testing. Hence, without modifying the reward, it would be negative when moving to the left.

Ideally, if some method could learn to walk in both directions, we should observe a testing plot where the return should become closer to zero as training proceeds. However, we do not see that behaviour in the two testing plots. We observe a lower average return in the testing plots, but it keeps growing as training proceeds.

The reward function has the term rewarding the agent for being alive, always positive, and the penalization term for taking too large actions, always negative. Thus, when walking to the left, we invert

the effect of those terms if we do not change the reward function. Nonetheless, if the robot could walk optimally to the left, the penalization term would not significantly change the episode return, and we would have to worry about the positive term, which can take at most 1,000. Thus, the highest negative return would be around -4,000.

In that case, the plot would not go to zero, but we would still see it decaying, which we do not see happen. In TD3, we have one run with a testing return of around 1,000, but it was always around that return. With JAD3, all runs keep increasing, meaning it did not learn to walk to the left either. We rendered what happened with both methods when walking to the left, and we mostly observed the robot falling or not moving because the legs conflicted, trying to move in different directions, stalling the robot.

We even reproduced the training conditions where we tried to walk to the left with our trick of rotating the agents and the torso information. When rendering those training episodes, both methods could walk to the right. Thus, once we undo that rotation in the testing episodes, the robot should also walk to the left. However, it could not, meaning our approach has an important flaw we overlooked.

Our next move was to partially undo the rotation in the testing episodes to see if some method could present a different behaviour. Keeping the torso information rotated did not work in any method, but keeping the agent rotation changed how the multi-agent method behaved. Instead of remaining still, the robot started walking to the right to turn around and go left. Then, instead of turning around again and walking in circles, it kept moving to the left as we intended.

However, even with that behaviour, it is not what we intended to achieve. The idea was for the multi-agent method to learn to control both pairs of legs with the same policy without needing that agent rotation. At least, it indicates that multi-agent methods allow learning some behaviour the single-agent cannot grasp.

We rendered how the ant learns to walk in each situation with both methods, and we include some frames from the multi-agent one in Appendix B and the GIFs for both in our GitHub repository. In Figure 63, we show how the multi-agent method learns to walk to the right (i.e. the initial task). It uses the red and blue legs to jump forward and the other to keep the balance and avoid falling.

In Figure 64, we show how it learns to walk to the left during training (i.e. faking the rotation). It is interesting because it learns to walk differently, using the four legs equally instead of the two diagonal it uses when walking to the right. Moreover, this other way of walking is more inefficient because it achieves lower return.

In Figure 65, we show some frames to depict how it learns to turn around and move to the left when we do not undo the agents' rotation. Once it turns around, it walks similar to what it walks to the right during training when rotating the agents and the torso information.

The only explanation we can think of right now is the policy has overfitted the legs each agent controls in each direction. If the direction flag indicates moving to the right, the first agent must handle the first and second legs. If it is left, the first agent must control the third and fourth. If we do not maintain this order, the robot does not move in any case, even when walking to the right like the original task, without rotating anything.

Moreover, we also need to keep which leg's torques each agent outputs first. When walking to the right, the first agent outputs the first leg's sub-actions. If we invert this order, we also face the same problem of the robot not moving.

In the end, we could not make it work as intended. After all, maybe it was easier to spend some

time tinkering with the reward function instead of the approach we followed. However, we were already short on time to change the approach. We decided to leave this demonstration inconclusive and hope to pick it up in another moment as future work.

8 Discussion

In this last section, we summarize our goals and extract conclusions from all the work we have done in this thesis, objectively describing how much we have accomplished our goals. Finally, we present different lines of future work we could follow, including what we had to discard during the development of this thesis.

8.1 Conclusions

The main problem we wanted to tackle in this thesis was solving tasks that a single-agent RL method could not because the action space was too large for the method to explore it enough to learn a good policy. With large action spaces, we usually refer to highly-dimensional actions, each dimension taking values in an infinite or finite set.

For this reason, our solution was leveraging multi-agent methods to decompose the initial single-agent problem into multiple agents and learn behaviours for each agent such that they act coordinated to achieve the original goal (i.e. maximize the accumulated single-agent reward).

We experimented with four environments, using three to demonstrate how we could leverage MARL methods to solve tasks that single-agent methods could not. All three had multi-dimensional actions, discrete or continuous, but the maximum dimension we tried was twenty if we consider how ManyAgent Swimmer 10x2 had two sub-actions per agent.

From the results of our experiments, we can confirm the flaws of single-agent methods in such tasks. In the Pistonball environment, DQN could learn a sub-optimal policy when we trained with ten pistons, but it failed to learn anything when increasing to fifteen. In the continuous version, TD3 failed to learn anything with 15 pistons.

Only in ManyAgent Swimmer 2x2, TD3 worked better because it had to generate a 4-dimensional vector. Moreover, the fully observable state had information that the observations did not have, and without it, learning the optimal policy was impossible. The information should have come through the mixer, but it could not. Even the people maintaining the environment added that information in the newer versions.

We still believe TD3 would learn the optimal policy faster than MARL methods if we added that information because of the low action dimensionality and the challenges of training multiple agents. We already saw how TD3 learned the same sub-policy faster when we trained it with eight environments. However, TD3 failed in the 10x2 version because the actions had too many dimensions for the method to learn something in that many training steps.

Thanks to all or most agents behaving identically, we demonstrated how we could train a shared network to control all instead of one per agent, accelerating learning. Moreover, thanks to training shared networks, we also demonstrated how we could learn a task and then use the learned weights as a starting point to train a more difficult version of the same task. With a single-agent method, we cannot reuse anything if we increase the action dimensionality because the network's output size changes.

Even if leveraging MARL methods can solve an initially single-agent task with high action dimensionality, we have to remember that we must meet some requirements to perform the decomposition

into several agents. All the environments we tried fulfilled them, but we would have needed another solution if they had not, like the methods we explained in the first subsection of the literature review, which maintained a single agent but applied different tricks.

One might ask why we have not trained any of the environments with those methods. We wanted to focus on MARL methods because they can take advantage of the agents behaving identically and reuse the learned policy with more agents. On the other hand, that other set of methods cannot. Moreover, those methods have limited applicability because they used DQN as the basis, and even if they could discretize the continuous actions, they would probably fail to learn some of our tasks.

Among MARL methods, we wanted to demonstrate the importance of centralizing training. In those three environments, we trained IQL, the naive approach that ignores the existence of the other agents. Our results demonstrated that IQL could not learn coordinated behaviour, failing to solve the more complex environments. Moreover, suffering from non-stationarity also caused it to obtain worse results.

We also wanted to demonstrate how methods learning to decompose the joint action-value function, learning implicit credit assignment, worked better than those not factorizing it. The latter type needs some explicit mechanism to learn each agent's contribution to the joint reward. However, some methods update the policy with the joint action-value function, the policy having to extract each agent contribution from a scalar value.

We only executed one that did not decompose the joint reward, MADDPG. The idea was to demonstrate how using such a method to solve any task would fail, which it did. Nonetheless, we might not have used the most adequate method, at least in the discrete environment, as we later found one that should work better. Our results demonstrate that MADDPG works worse than methods decomposing the joint action values.

We used MADDPG because it was originally a method working with continuous actions, and we did not find another one that could work with such actions. Nonetheless, the authors somehow created a discrete version. With that, our fourth objective was to focus our thesis on environments with continuous actions.

When we researched the state of the art of MARL, we mostly found methods working with discrete actions. Even if they could work with continuous ones, their authors never demonstrated it with results. And the problem was worse when specifically searching for continuous methods, as we could only find very few.

For this reason, if we could not find what we wanted, our solution was to create it ourselves, giving us a fifth objective. We created JAD3 based on FACMAC, which we wanted to improve because we did not agree with some aspects, like not leveraging the decomposition to update the actor and using DDPG as the basis when there are better options.

Therefore, when experimenting with the continuous environments, we also wanted to test if JAD3 could surpass FACMAC and determine which action-value function we should use when updating the actor: the joint or the individuals we learn through decomposition. We expected the individual ones to work better, and our results proved it with a significant difference. It learned faster and oscillated less.

When comparing it with FACMAC, JAD3 obtained better results, especially the version updating the actors with the individual action values, which presented high differences. Nonetheless, once both methods grasped the better policy, FACMAC could catch up with JAD3. It would have been interesting to test more environments the authors of FACMAC had not tried and compared them there.

With the fourth environment, we wanted to demonstrate how MARL methods could learn general policies that could solve at the same time similar tasks in an environment. On the other hand, a single-agent method should fail to learn such general policies, having to learn them separately instead.

We experimented with TD3 and our new method. However, we did not obtain conclusive results that could help us demonstrate that fact. We followed an approach we thought would be easy, but it was not. As each experiment lasted almost two days, and we were already short of time, we had to leave it unfinished.

8.2 Future Work

First, we would like to continue experimenting with the Ant environment to see if we could learn a general policy with the MARL method that could walk in the two directions. Hopefully, with the single-agent method, it should not learn such general policy, only learning to walk in one direction. With those results of the Ant turning around when walking to the left using the MARL policy, we believe we could achieve what we intended.

Second, we would test TransfQMIX further to see if we could achieve better results in the Pistonball environments because we still believe we should have obtained better ones in such a simple environment. Moreover, we would also like to experiment with the current version of the ManyAgent Swimmer Farama maintains to see if the MARL methods can learn the optimal policy.

Third, we would like to extend our experimentation and test more environments, especially with continuous actions. Moreover, we would like to try more MARL methods, especially those not factorizing the joint function, to demonstrate how they work worse than those decomposing it.

Fourth, we would test JAD3 in more continuous environments to demonstrate further how it improves FACMAC. Moreover, we would like to try using another method as a basis, as there are other options besides TD3, for example, SAC. It is true that the multi-agent version of SAC already exists, but the authors did not demonstrate how it could work with continuous environments.

Fifth, we would continue two other contributions we wanted to make at the beginning of this thesis that we had to discard because we encountered some difficulties we could not solve. We explained those contributions in the progress report we delivered some time ago, so we will briefly mention them again.

The first one was extending TransfQMIX to work with continuous actions. We worked with its author, a former MAI student, and created the first version that worked decently in the Pistonball environment, but we were not satisfied. However, as we were working on multiple experiments, and the author had to attend to other important matters, we left it aside.

The second contribution was solving an environment resembling a real-world problem instead of the usual benchmark tasks. It featured a telescope from the adaptive optics field (AO) that we had to learn to control. We would not directly work with the real deal but use a simulator another student working with our supervisor created. He obtained good results with a single-agent method, and we wanted to leverage multi-agent methods instead.

The idea was to prove the advantages of MARL methods to speed up learning and demonstrate how they could learn more general policies. Without entering into detail, the environment fulfilled the requirements we explained before. However, we could not make it learn with the MARL methods, and we had to switch to the Ant environment we explained.

References

[1] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2023. URL: https://www.marl-book.com.

[2] Marc G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *J. Artif. Int. Res.* 47.1 (May 2013), pp. 253–279. ISSN: 1076-9757.

[3] Richard Bellman. "A Markovian Decision Process". In: *Indiana Univ. Math. J.* 6 (4 1957), pp. 679–684. ISSN: 0022-2518.

[4] Daniel S. Bernstein, Shlomo Zilberstein, and Neil Immerman. "The Complexity of Decentralized Control of Markov Decision Processes". In: *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*. UAI'00. Stanford, California: Morgan Kaufmann Publishers Inc., 2000, pp. 32–37. ISBN: 1558607099.

[5] Petros Christodoulou. "Soft Actor-Critic for Discrete Action Settings". In: *CoRR* abs/1910.07207 (2019). arXiv: 1910.07207. URL: http://arxiv.org/abs/1910.07207.

[6] Rémi Coulom. "Reinforcement Learning Using Neural Networks, with Applications to Motor Control". PhD thesis. Institut National Polytechnique de Grenoble, 2002.

[7] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[8] Gabriel Dulac-Arnold et al. "Deep Reinforcement Learning in Large Discrete Action Spaces". In: *CoRR* abs/1512.07679 (2015). arXiv: 1512.07679. URL: http://arxiv.org/abs/1512.07679.

[9] Jakob N. Foerster et al. "Counterfactual Multi-Agent Policy Gradients". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI'18/IAAI'18/EAAI'18. New Orleans, Louisiana, USA: AAAI Press, 2018. ISBN: 978-1-57735-800-8.

[10] Jakob N. Foerster et al. "Counterfactual Multi-Agent Policy Gradients". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI'18/IAAI'18/EAAI'18. New Orleans, Louisiana, USA: AAAI Press, 2018. ISBN: 978-1-57735-800-8.

[11] Maël Franceschetti et al. *Making Reinforcement Learning Work on Swimmer*. 2022. arXiv: 2208.07587 [cs.LG].

[12] Wei Fu et al. "Revisiting Some Common Practices in Cooperative Multi-Agent Reinforcement Learning". In: *Proceedings of the 39th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri et al. Vol. 162. Proceedings of Machine Learning Research. PMLR, July 2022, pp. 6863–6877. URL: https://proceedings.mlr.press/v162/fu22d.html.

[13] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, July 2018, pp. 1587–1596. URL: https://proceedings.mlr.press/v80/fujimoto18a.html.

[14] Matteo Gallici, Mario Martin, and Ivan Masmitja. *TransfQMix: Transformers for Leveraging the Graph Structure of Multi-Agent Reinforcement Learning Problems*. 2023. arXiv: 2301.05334 [cs.LG].

[15] David Ha, Andrew M. Dai, and Quoc V. Le. "HyperNetworks". In: *CoRR* abs/1609.09106 (2016). arXiv: 1609.09106. URL: http://arxiv.org/abs/1609.09106.

[16] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, July 2018, pp. 1861–1870. URL: https://proceedings.mlr.press/v80/haarnoja18b.html.

[17] Matthew J. Hausknecht and Peter Stone. "Deep Recurrent Q-Learning for Partially Observable MDPs". In: *CoRR* abs/1507.06527 (2015). arXiv: 1507.06527. URL: http://arxiv.org/abs/1507.06527.

[18] Eric Jang, Shixiang Gu, and Ben Poole. *Categorical Reparameterization with Gumbel-Softmax*. 2017. arXiv: 1611.01144 [stat.ML].

[19] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial Intelligence* 101.1 (1998), pp. 99–134. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(98)00023-X. URL: https://www.sciencedirect.com/science/article/pii/S000437029800023X.

[20] Dmitry Kalashnikov et al. "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation". In: *CoRR* abs/1806.10293 (2018). arXiv: 1806.10293. URL: http://arxiv.org/abs/1806.10293.

[21] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: 1312.6114 [stat.ML].

[22] Florian Köpf et al. "Deep Decentralized Reinforcement Learning for Cooperative Control". In: *CoRR* abs/1910.13196 (2019). arXiv: 1910.13196. URL: http://arxiv.org/abs/1910.13196.

[23] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning." In: *ICLR*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

[24] Ryan Lowe et al. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6382–6393. ISBN: 9781510860964.

[25] Xueguang Lyu et al. "Contrasting Centralized and Decentralized Critics in Multi-Agent Reinforcement Learning". In: *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '21. Virtual Event, United Kingdom: International Foundation for Autonomous Agents and Multiagent Systems, 2021, pp. 844–852. ISBN: 9781450383073.

[26] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[27] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). NIPS Deep Learning Workshop 2013. URL: http://arxiv.org/abs/1312.5602.

[28] Marius Muja and David G. Lowe. "Scalable Nearest Neighbor Algorithms for High Dimensional Data". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11 (2014), pp. 2227–2240. DOI: 10.1109/TPAMI.2014.2321376.

[29] Frans A. Oliehoek, Matthijs T. J. Spaan, and Nikos Vlassis. "Optimal and Approximate Q-value Functions for Decentralized POMDPs". In: *CoRR* abs/1111.0062 (2011). arXiv: 1111.0062. URL: http://arxiv.org/abs/1111.0062.

[30] Georgios Papoudakis et al. "Benchmarking Multi-Agent Deep Reinforcement Learning Algorithms in Cooperative Tasks". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS)*. 2021. URL: http://arxiv.org/abs/2006.07869.

[31] Bei Peng et al. "FACMAC: Factored Multi-Agent Centralised Policy Gradients". In: *Advances in Neural Information Processing Systems*. Ed. by A. Beygelzimer et al. 2021. URL: https://openreview.net/forum?id=wZYWwJvkneF.

[32] Rafael Pina et al. "Residual Q-Networks for Value Function Factorizing in Multiagent Reinforcement Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* (2022), pp. 1–11. DOI: 10.1109/TNNLS.2022.3183865.

[33] Yuan Pu et al. "Decomposed Soft Actor-Critic Method for Cooperative Multi-Agent Reinforcement Learning". In: *CoRR* abs/2104.06655 (2021). arXiv: 2104.06655. URL: https://arxiv.org/abs/2104.06655.

[34] Tabish Rashid et al. "QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning". In: *ArXiv* abs/1803.11485 (2018).

[35] Mikayel Samvelyan et al. "The StarCraft Multi-Agent Challenge". In: *CoRR* abs/1902.04043 (2019). arXiv: 1902.04043. URL: http://arxiv.org/abs/1902.04043.

[36] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1506.02438.

[37] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". en. In: *arXiv:1506.02438 [cs]* (Oct. 2018). arXiv: 1506.02438. URL: http://arxiv.org/abs/1506.02438 (visited on 12/31/2019).

[38] Kyunghwan Son et al. "Qtran: Learning to factorize with transformation for cooperative multi-agent reinforcement learning". In: *International conference on machine learning*. PMLR. 2019, pp. 5887–5896.

[39] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[40] Ming Tan. "Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents". In: *Readings in Agents*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 487–494. ISBN: 1558604952.

[41] Arash Tavakoli, Fabio Pardo, and Petar Kormushev. "Action Branching Architectures for Deep Reinforcement Learning". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI'18/IAAI'18/EAAI'18. New Orleans, Louisiana, USA: AAAI Press, 2018. ISBN: 978-1-57735-800-8.

[42] Justin K. Terry et al. "Parameter Sharing For Heterogeneous Agents in Multi-Agent Reinforcement Learning". In: *CoRR* abs/2005.13625 (2020). arXiv: 2005.13625. URL: https://arxiv.org/abs/2005.13625.

[43] Sebastian Thrun and Anton Schwartz. "Issues in Using Function Approximation for Reinforcement Learning". In: *Proceedings of the 1993 Connectionist Models Summer School*. Ed. by Michael Mozer et al. Lawrence Erlbaum, 1993, pp. 255–263. URL: http://www.ri.cmu.edu/pub_files/pub1/thrun_sebastian_1993_1/thrun_sebastian_1993_1.pdf.

[44] Callum Rhys Tilbury, Filippos Christianos, and Stefano V. Albrecht. *Revisiting the Gumbel-Softmax in MADDPG*. 2023. arXiv: 2302.11793 [cs.LG].

[45] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[46] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[47] Jianhao Wang et al. "QPLEX: Duplex Dueling Multi-Agent Q-Learning". In: *CoRR* abs/2008.01062 (2020). arXiv: 2008.01062. URL: https://arxiv.org/abs/2008.01062.

[48] Yihan Wang et al. "Off-Policy Multi-Agent Decomposed Policy Gradients". In: *CoRR* abs/2007.12322 (2020). arXiv: 2007.12322. URL: https://arxiv.org/abs/2007.12322.

[49] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 1995–2003.

[50] Tom Van de Wiele et al. "Q-Learning in enormous action spaces via amortized approximate maximization". In: *CoRR* abs/2001.08116 (2020). arXiv: 2001.08116. URL: https://arxiv.org/abs/2001.08116.

[51] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[52] DAVID H. WOLPERT and KAGAN TUMER. "OPTIMAL PAYOFF FUNCTIONS FOR MEMBERS OF COLLECTIVES". In: *Advances in Complex Systems* 04.02n03 (2001), pp. 265–279. DOI: 10.1142/S0219525901000188. eprint: https://doi.org/10.1142/S0219525901000188. URL: https://doi.org/10.1142/S0219525901000188.

[53] Tom Zahavy et al. "Learn What Not to Learn: Action Elimination with Deep Reinforcement Learning". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/645098b086d2f9e1e0e939c27f9f2d6f-Paper.pdf.

[54] Meng Zhou et al. "Learning Implicit Credit Assignment for Cooperative Multi-Agent Reinforcement Learning". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 11853–11864. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/8977ecbb8cb82d77fb091c7a7f186163-Paper.pdf.

A Hyperparameters

In this first appendix, we want to explain the different hyperparameters we can change in the methods we have trained. Moreover, we will also mention their default value, as those hyperparameters we have not discussed when explaining the results mean they had that default value. We present them below, ordering them depending on their function:

- Exploring the action space:
  - In the discrete case, we use ϵ-greedy to explore the action space, decreasing the ϵ as training proceeds. Hence, we have three parameters:
    - Initial ϵ value, which mostly takes the value of 1.
    - Final ϵ value, which always takes the value of 0.05.
    - Number of steps to linearly decay ϵ, which each method sets its value.
  - In the continuous case, we sample noise from a Gaussian to modify the action. We set the μ to 0 but change σ, decreasing it linearly as training proceeds. Depending on the method, we have three or seven parameters. In most cases, we have the following three:
    - Initial σ, which each method sets its value.
    - Final σ, which each method also sets its value, but we mostly set it to 0.05.
    - Number of steps to linearly decay σ. This parameter is the same as the number of training steps, unlike ϵ-greedy, which reduces ϵ to the minimum before training ends.

    In TD3 and JAD3, we also add Gaussian noise to the target actions, so we have the same three parameters with the same way of setting their values. However, we clip the noise we add to those actions, so we have an extra parameter defining the magnitude of the maximum noise. We have used 0.5 in all but one case.
- Updating the networks:
  - We have one or two parameters to specify the learning rate we use to update each network, leveraging Adam. In actor-critic methods, we have different values for the actor and the critic (or critic+mixer in MARL). The default value is 0.001 unless we specify otherwise.
  - Similarly, we have one or two parameters to specify the maximum gradient norm we allow before clipping it. Each method sets the value differently depending on the environment. However, we mainly use 0.5.
  - We update all the target networks using polyak average, having a parameter to set ρ. We have always used 0.005.
  - In TD3 and JAD3, there is a parameter to control how frequently we update the actors and target networks. TD3 always updates them every two updates. However, in JAD3, depending on the number of parallel environments and training steps, we found it better to update them each time.
- Computing the loss functions:
  - One parameter specifies the discount factor γ, which mostly takes the value of 0.99, only changing it when we trained the ManyAgent Swimmer environment with TD3, where we set it to 0.9999.
  - In JAD3, there is a parameter to change how we update the actors, using the joint action-value function the mixer approximates or the individual ones learn through decomposition.
  - Some continuous methods have a parameter to allow a regularization term in the actor loss to avoid taking too large actions. Nonetheless, as most environments already have such a term in the reward function, we do not use it.

- Generating the inputs to the networks (only applies to MARL methods):
  - Most methods have a parameter to generate agent identifiers to add as input. TransfQMIX does not have it, as we add that information through the entity features (i.e. changing the environment definition). We have added them when experimenting with JAD3 in the ManyAgent Swimmer but not in the other environments.
  - The discrete methods, which use recurrent layers, have a parameter to add the previous action together with the observation as input. We only added them when training the discrete Pistonball environment.
  - The continuous methods, which do not use recurrent layers, have one parameter to control how many previous observation-action transitions we add as input, together with the current observation. We have not used that parameter in any of our experiments.
- Creating the networks:
  - Value-based methods have a parameter to select the architecture of the action-value network.
  - Actor-critic methods have a parameter to select the actor's architecture and another to select the critic's.
  - MARL methods decomposing Q_{tot} have a parameter to select the type of decomposition. However, most methods, even the continuous ones, use QMIX's factorization. The only one that uses a different one is VDN itself.
  - We have similar parameters to determine the number of units per layer in each type of network. In MARL methods factorizing Q_{tot}, we also set the number in the mixer network and the hypernetworks.

  We have tried changing the size of the networks in our experiments. However, the architecture is mostly bound to the method, especially when using recurrent layers or not. We can change the number of networks we train in the multi-agent methods. However, we do not recommend changing the type of layers the networks use.

  If not specified otherwise, the default units per layer are as follows:
  - In multi-agent methods, the action-value networks and the actor and critic networks use 64 units per layer. The mixer and the hypernetworks use 32 units.
  - In single-agent methods, we have set 64 units in the discrete environments and 256 in the continuous ones.
- Other parameters:
  - One parameter determines the batch size. We used a batch size of 20 when sampling full episodes, while we used a batch of 100 when sampling transitions. We tried different batch sizes, but the change was not significant.
  - One parameter determines the buffer size. We used a buffer size of 1,000 episodes when sampling episodes, 500,000 or 1,000,000 transitions when sampling transitions in the MuJoCo environments, and 50,000 or 100,000 in the continuous Pistonball.
  - One parameter determines the number of parallel environments. In most cases, we trained with one except when training with FACMAC and our method in the ManyAgent Swimmer.

B Simulations

In this appendix, we have included the frames we obtained by rendering the different environments using the policies learned by different methods. The frames correspond to testing episodes, meaning we execute the actions the policy would output without any perturbation.

We decided to include the frames to view what is happening directly in the same document. Nonetheless, as we have explained before, we have uploaded the rendering of those episodes as GIFs in our GitHub repository, which we link in Appendix C, even for those methods or environments we do not present here.

B.1 Pistonball

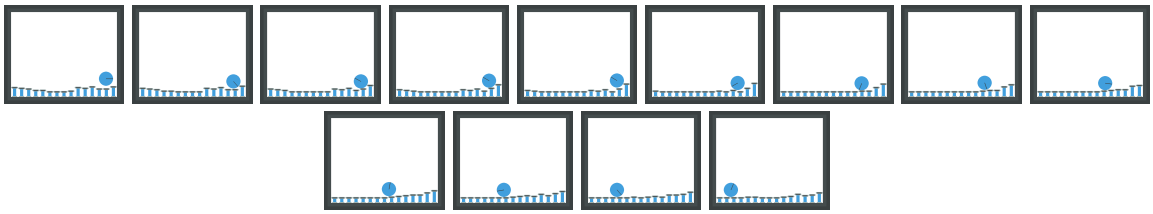


Figure 53: Pistonball discrete - Rendering - QMIX shared

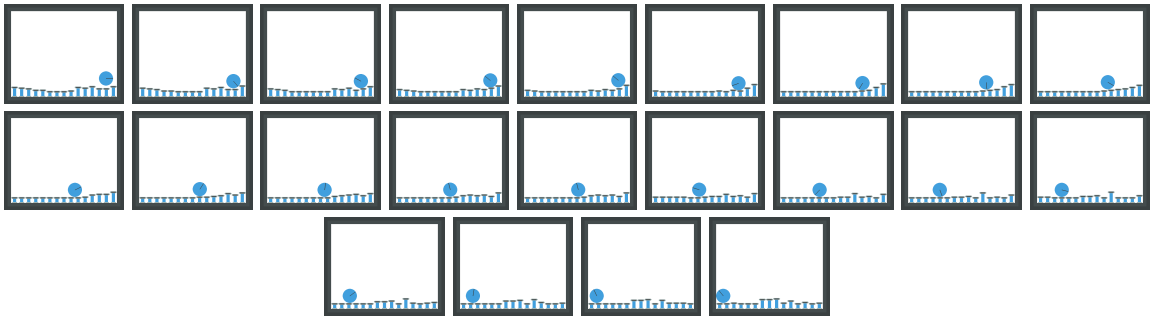


Figure 54: Pistonball discrete - Rendering - IQL

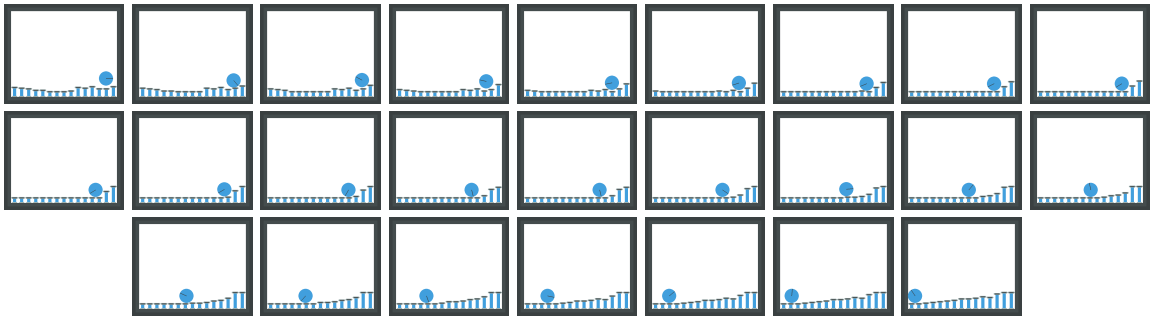


Figure 55: Pistonball discrete - Rendering - TransfQMIX

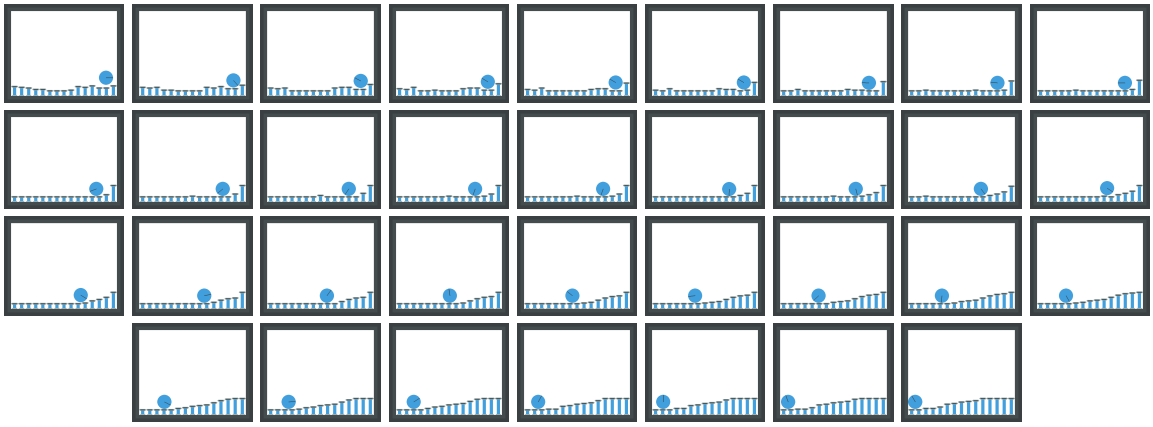


Figure 56: Pistonball discrete - Rendering - MADDPG

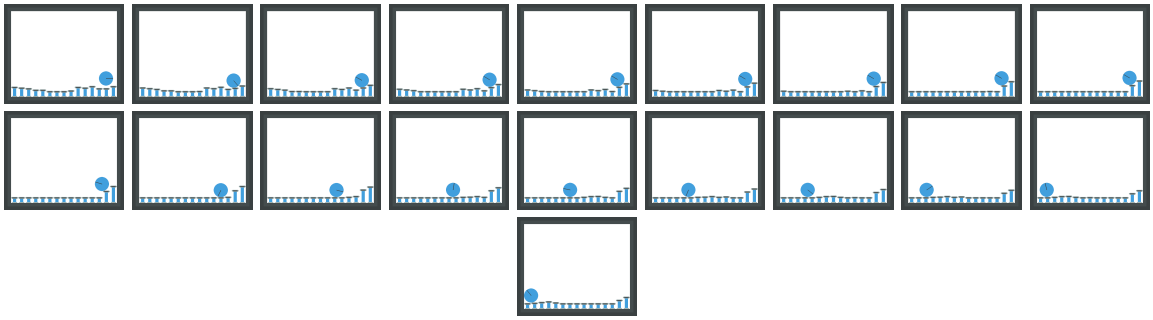


Figure 57: Pistonball continuous - Rendering - JAD3 - Updating the actor with $[Q_a]_{a=1}^{N}$

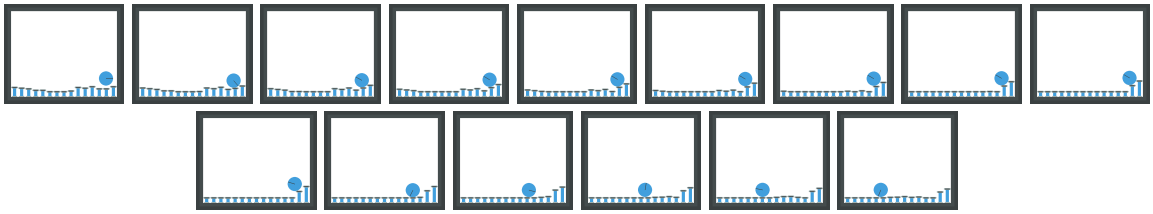


Figure 58: Pistonball continuous - Rendering - JAD3 - Updating the actor with Q_{tot}

Figure 59: Pistonball continuous - Rendering - IQL

B.2 ManyAgent Swimmer 2x2

We changed the floor texture in the frames we present below from the checkerboard pattern to a flat colour because looking at them made us dizzy. Nonetheless, in the GIFs, we left the original texture. Moreover, in all the renderings we present, the first segment (i.e. the non-controllable one) is below the others. However, some policies learned to walk such that it was over, the Swimmer swimming inverted. Some of the GIFs we have uploaded to our repository walk like that.

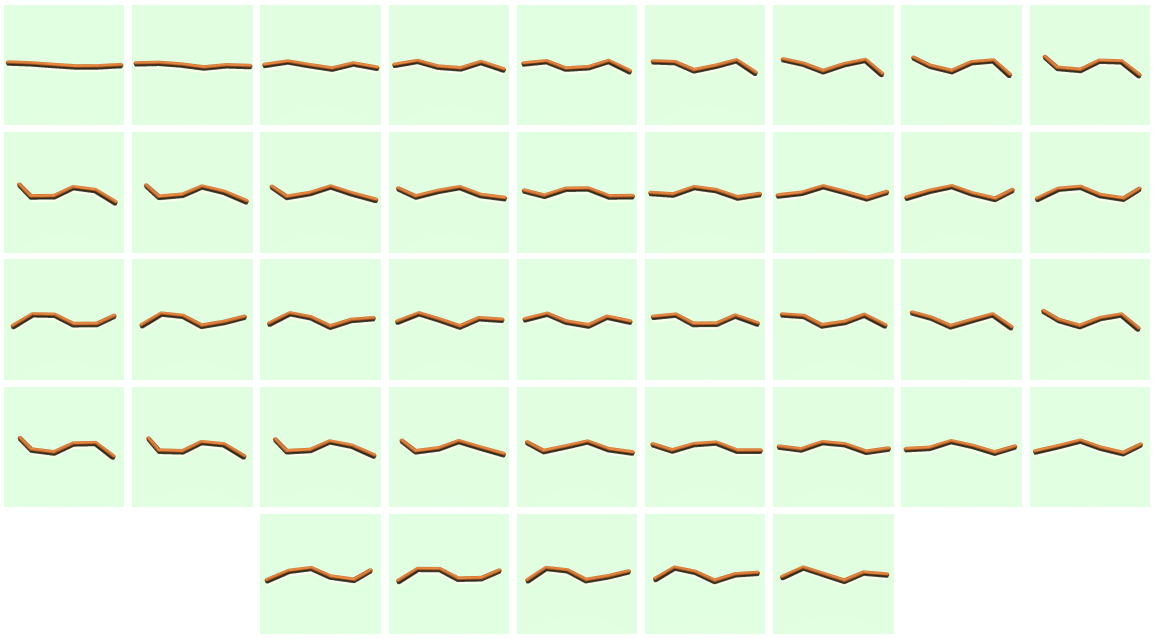


Figure 60: ManyAgent Swimmer 2x2 - Rendering - TD3 - Optimal behaviour (around 460 return)

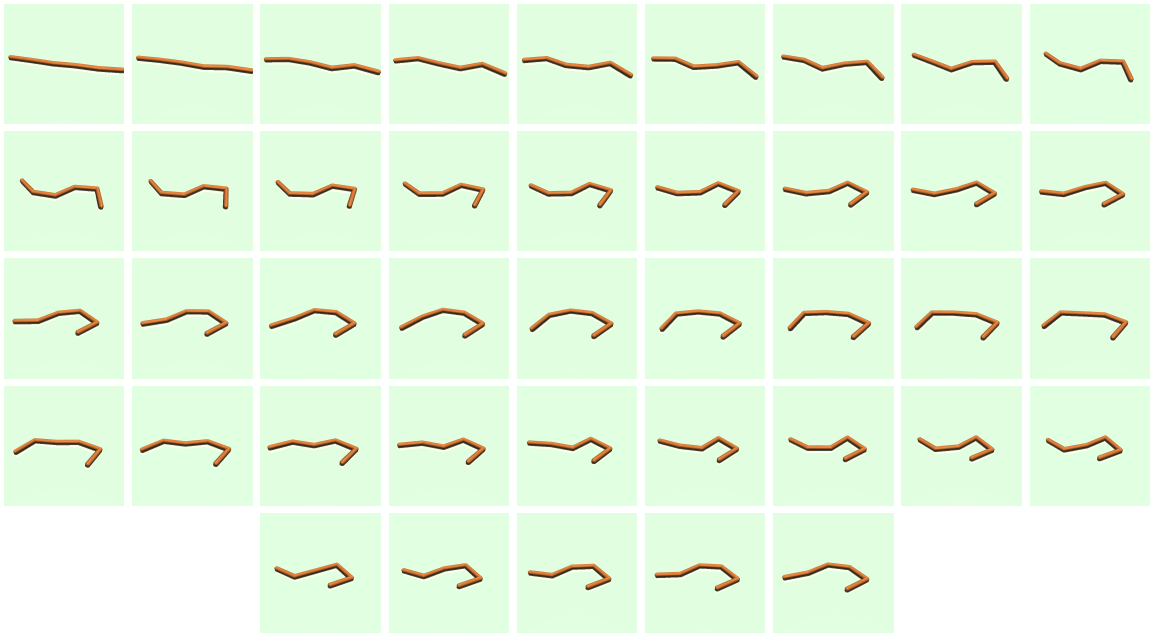


Figure 61: ManyAgent Swimmer 2x2 - Rendering - JAD3 - Sub-optimal behaviour (around 250 return)

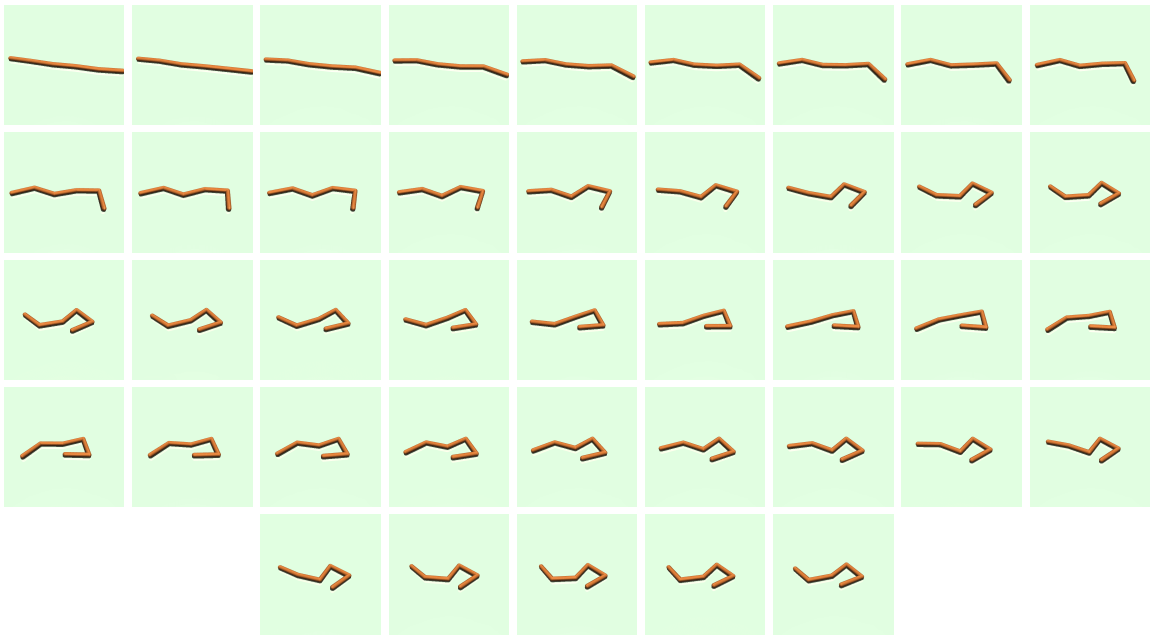


Figure 62: ManyAgent Swimmer 2x2 - Rendering - JAD3 - Sub-optimal behaviour (200 return)

B.3 Ant

As in the ManyAgent Swimmer frames, we also changed the floor texture to a flat colour. However, in the GIFs we uploaded to our GitHub, including those from TD3, we left the original one.

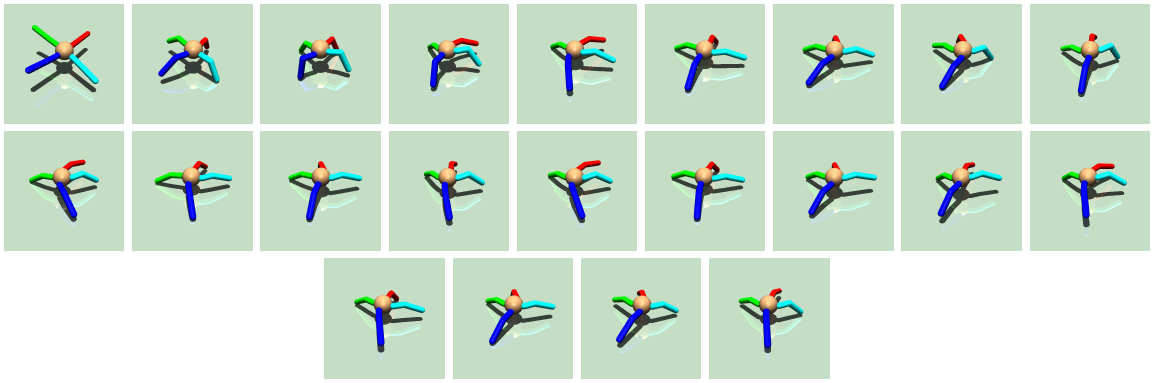


Figure 63: Ant - Rendering - JAD3 - Walking to the right (original task)

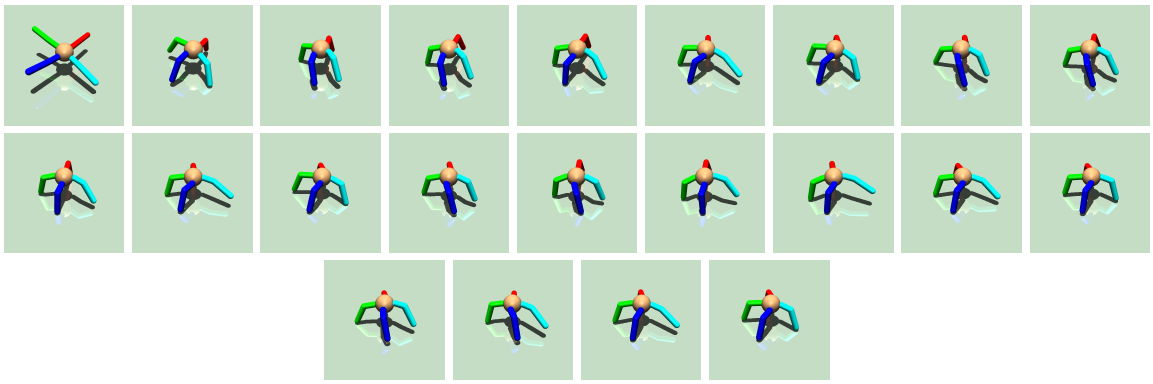


Figure 64: Ant - Rendering - JAD3 - Walking to the left during training (fake rotation)

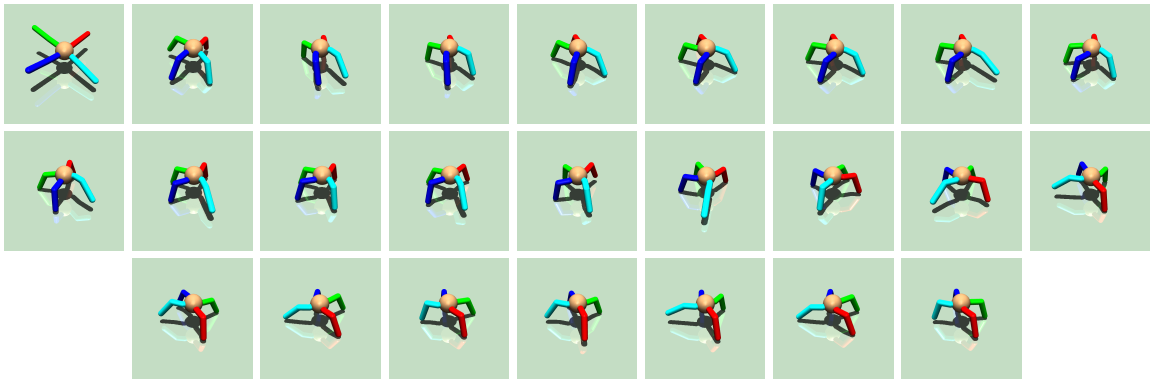


Figure 65: Ant - Rendering - JAD3 - Walking to the left (agents rotated)

C GitHub Repository

In this last appendix, we want to explain the contents of our GitHub repository, where anyone can find our code and try to reproduce our results. The link to the repository is https://github.com/QuimMarset/TFM, and we can find five main elements:

- A README file explaining how to execute our code and what you can find in the repository.
- A *requirements* folder with two TXT files with the Python libraries (and their versions) we have used to perform our experiments. We leveraged the BSC cluster and needed some libraries they had not installed. Hence, they created two images with different Python libraries, one to execute the Pistonball environment and another to run the MuJoCo ones. They could not fuse them in one, and each image has different versions of the same libraries, like PyTorch.
- A *src* folder with all the Python source code. Inside *src/config*, a Markdown file defines the hyperparameters explained in Appendix A with the names used in the code.
- A *gifs* folder with the rendered testing episodes of the different environments and methods saved as GIFs. We have at least included one per method and environment, but not all if we have several for the same. For example, with ManyAgent Swimmer 2x2 and JAD3, we have only included the best one with each update mechanism. Moreover, we have cut the GIFs' duration in the environments with the longest episodes as they would take too much space otherwise.
- A *configs* folder with the best configurations we found with each method and environment.